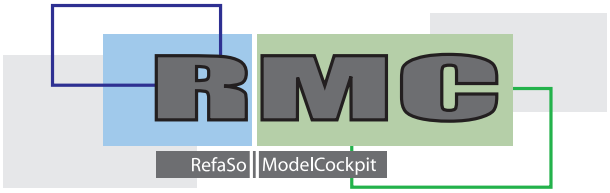
Abschlussdokumentation

Abschlussdokumentation

der Projektgruppe RefaSo & ModelCockpit

21. April 2008



Universität Paderborn, Fachgruppen Prof. Dr. Heike Wehrheim und
Prof. Dr. Gregor Engels

Mitglieder der Projektgruppe:

André Kemena, Bernhard Dietrich, Christoph Oberhokamp, Martin Kleine, Mathias Raacke, Meik Piepmeyer, Peter Winkelhane, Rudolf Braun, Steffen Dohle, Tobias Friedrich, Yi Tan

Universität Paderborn
Erste Auflage: 6 Exemplare

Diese Dokumentation wurde mit Hilfe von LaTeX gesetzt.

...der Semantik,
die uns stets viel Freud'
und Leid beschert hat.

Inhaltsverzeichnis

I Konzeptionelle Sicht 21

1 Einleitung 23
- 1.1 Motivation . . . 23
- 1.2 Inhaltliche Übersicht . . . 24

2 Konzepte 25
- 2.1 Die Sprache UML/Z . . . 25
- 2.2 Qualitätskreislauf . . . 26
  - 2.2.1 Modellieren der Qualität . . . 28
  - 2.2.2 Messen der Qualität . . . 29
  - 2.2.3 Verbessern der Qualität . . . 30

II Anwenderorientierte Sicht 31

3 Installationsanleitung 33

4 Benutzerdokumentation 35
- 4.1 Einleitung . . . 35
- 4.2 Komponenten . . . 35
  - 4.2.1 Projekte . . . 36
  - 4.2.2 Editor . . . 36
    - 4.2.2.1 Diagramme anlegen . . . 36
    - 4.2.2.2 Klassendiagramme . . . 40
    - 4.2.2.3 Zustandsdiagramme . . . 53
  - 4.2.3 Modellqualitätsplan . . . 58
    - 4.2.3.1 MQP anlegen . . . 58
    - 4.2.3.2 Der MQP-Editor . . . 60
    - 4.2.3.3 MQP-Kontextmodell . . . 61
    - 4.2.3.4 MQP-Informationsbedürfnismodell . . . 65
    - 4.2.3.5 MQP-Qualitätsmodell . . . 68
    - 4.2.3.6 MQP-Messmodell . . . 70
    - 4.2.3.7 MQP-Präsentationsmodell . . . 73
  - 4.2.4 Diagnose . . . 76
    - 4.2.4.1 Benutzung der Diagnose . . . 76
    - 4.2.4.2 Konfigurationsdatei . . . 77
  - 4.2.5 Refactoring . . . 79

4.2.5.1 Ausführen von Refactorings . . . 79
4.2.5.2 Konfigurationsdatei . . . 82
4.3 FAQ . . . 89
4.3.1 Allgemeine Probleme . . . 89
4.3.2 Projekte . . . 89
4.3.3 Editor . . . 89
4.3.4 Modellqualitätsplan . . . 90
4.3.5 Diagnose . . . 90
4.3.6 Refactoring . . . 90

5 Tutorial **91**
5.1 Einleitung . . . 91
5.2 Durchgehendes Beispiel . . . 91
5.2.1 Editor . . . 91
5.2.1.1 Projekt anlegen . . . 92
5.2.1.2 Datentypen erstellen . . . 95
5.2.1.3 Klassen anlegen . . . 103
5.2.1.4 Beziehungen der Klassen modellieren . . . 117
5.2.2 Modellqualitätsplan . . . 119
5.2.2.1 Einen neuen Modellqualitätsplan anlegen . . . 121
5.2.2.2 Ein neues Kontextmodell erstellen . . . 123
5.2.2.3 Ein neues Informationsbedürfnismodell erstellen . . . 124
5.2.2.4 Ein neues Qualitätsmodell erstellen . . . 128
5.2.2.5 Ein neues Messmodell erstellen . . . 128
5.2.2.6 Das Präsentationsmodell verwenden . . . 131
5.2.3 Diagnose . . . 134
5.2.4 Refactoring . . . 135

III Entwicklerorientierte Sicht **141**

6 Architekturdokumentation **143**
6.1 Komponentenübersicht . . . 143
6.2 Schnittstellen . . . 147
6.2.1 IModel . . . 147
6.2.2 IRefactoringEditor . . . 147
6.2.3 IRefactoringDiagnostics . . . 148
6.2.4 IOCL . . . 149
6.2.5 IMQP . . . 149
6.2.6 IZParser . . . 150
6.3 Komponenten . . . 153
6.3.1 UML/Z-Model . . . 153
6.3.1.1 Ergänzung der UML2 zu UML/Z . . . 155
6.3.1.2 Anbindung des Z-Parsers an das Modell . . . 158
6.3.2 UML/Z-Editor . . . 158
6.3.2.1 Klassendiagrammeditor . . . 160
6.3.2.2 Zustandsdiagrammeditor . . . 160

6.3.3 ModelQualityPlan . . . 160
6.3.3.1 Struktur . . . 162
6.3.3.2 Metamodell . . . 162
6.3.3.3 Teilkomponente MQP-GUI . . . 168
6.3.3.4 Teilkomponente MQP-Measurement . . . 169
6.3.4 Diagnostics . . . 172
6.3.4.1 Struktur . . . 172
6.3.4.2 Verhalten . . . 176
6.3.5 Refactoring . . . 180
6.3.5.1 Struktur . . . 180
6.3.5.2 Verhalten . . . 185
6.3.6 OCLWrapper . . . 189
6.3.7 CZTWrapper . . . 193

7 Entwicklerdokumentation 197
7.1 Installation der Entwicklungsumgebung . . . 197
7.1.1 Voraussetzungen . . . 197
7.1.2 Installation von Eclipse und der Plugins . . . 197
7.1.3 Importieren des RMC-Quellcodes . . . 199
7.1.4 Erstellen von RMC über das Ant-Script . . . 200
7.1.5 Voraussetzungen . . . 201
7.1.6 Build-Tasks . . . 201
7.1.7 Konfiguration des Build-Scripts . . . 201
7.2 Komponenten . . . 202
7.2.1 Modell . . . 203
7.2.1.1 Allgemeines . . . 203
7.2.1.2 Durchgeführte Erweiterungen . . . 204
7.2.1.3 Implementierung neuer Funktionen . . . 205
7.2.2 Editor . . . 210
7.2.2.1 Allgemeines . . . 210
7.2.2.2 Durchgeführte Erweiterungen . . . 214
7.2.2.3 Implementierung neuer Funktionen . . . 220
7.2.3 Modellqualitätsplan . . . 223
7.2.3.1 Änderungen am Modell . . . 223
7.2.3.2 GUI des Editors . . . 224
7.2.3.3 Zugriff auf das Modell . . . 228
7.2.4 Diagnose . . . 230
7.2.4.1 Erweiterung der Konfigurationsdatei . . . 231
7.2.4.2 Anlegen einer neuen Normalisierung . . . 233
7.2.4.3 Anlegen eines neuen Normalisierungstypes . . . 233
7.2.4.4 Mögliche Erweiterungen . . . 234
7.2.5 Refactoring . . . 234
7.2.5.1 Aufbau der Grammatik . . . 234
7.2.5.2 Abhängigkeiten der Grammatik . . . 238
7.2.5.3 Decisions anlegen . . . 240

8 API-Dokumentation 243

8.1 Javadoc . . . 243
8.2 UML/Z-Dokumentation für den Zugriff mit OCL . . . 243

9 Resümee 245
9.1 Entstehung und Ziele der Projektgruppe . . . 245
9.2 Vergleich von anderen Werkzeugen . . . 247
9.3 Besonderheiten unserer Lösung . . . 248
9.3.1 UML-ähnliche Modellierung mit UML/Z . . . 248
9.3.2 Modellierung und Messung von Qualität . . . 249
9.3.3 Diagnose der Messergebnisse . . . 249
9.3.4 Konfigurierbare Refactorings . . . 250
9.4 Vergleich des Ergebnisses mit der Zielsetzung . . . 250
9.5 Ausblick . . . 250

A Unsere Vorgehensweise 257
A.1 Projektorganisation . . . 257
A.2 Themen der Seminarphase . . . 258
A.3 Verwendete Werkzeuge . . . 260
A.4 Hinweise für zukünftige Projektgruppen . . . 261
A.4.1 Zeitmanagement . . . 261
A.4.2 Kein Projektleiter . . . 261
A.4.3 Zusammenarbeit vor Ort . . . 262
A.4.4 Entscheidungsfindung mit Hilfe eines Kriterienkatalogs . . . 262
A.4.5 Zwei Betreuer . . . 262
A.4.6 Spezialisten für einzelne Themen vermeiden . . . 262
A.4.7 Freizeitaktivitäten . . . 262

B OCL-Messungen 263
B.1 Scale of measure results . . . 263
B.2 Auxiliary Functions . . . 264
B.2.1 Property level functions . . . 264
B.2.2 Class level functions . . . 264
B.2.3 Model level functions . . . 270
B.2.4 StateMachine level functions . . . 271
B.3 Base Measures . . . 272
B.3.1 Unit of measure is Class . . . 272
B.3.2 Unit of measure is Model . . . 281
B.3.3 Unit of measure is StateMachine . . . 282
B.4 Derived measures . . . 283
B.4.1 Unit of measure is Class . . . 283
B.4.2 Unit of measure is Model . . . 286
B.5 Indicators . . . 294
B.5.1 Unit of measure is Class . . . 294
B.5.2 Unit of measure is Model . . . 295

C Lizenz 297

Abbildungsverzeichnis

2.1 Qualitätskreislauf 27

4.1 Projekt-Wizard: Startseite 36
4.2 Projekt-Wizard: 2. Seite 37
4.3 Projekt-Wizard: Diagrammtyp 38
4.4 Projekt-Wizard: Diagrammname 38
4.5 Projekt-Wizard: Modell festlegen 39
4.6 Editor: Diagramm auf Modell anlegen 39
4.7 Projekt-Wizard: Diagrammname 40
4.8 Projekt-Wizard: Elemtent für Diagramm wählen 41
4.9 Klassendiagrammeditor: Übersicht 41
4.10 Klassendiagrammeditor: Outline 42
4.11 Klassendiagrammeditor: Popup des Canvas 45
4.12 Klassendiagrammeditor: PropertyView 47
4.13 Klassendiagrammeditor: Z-Editor 48
4.14 Zustandsdiagrammeditor: Übersicht 53
4.15 Zustandsdiagrammeditor: Popup des Canvas 56
4.16 Zustandsdiagrammeditor: Transitionen 57
4.17 MQP-Wizard: MQP auswählen 58
4.18 MQP-Wizard: MQP-Name 59
4.19 MQP-Wizard: MQP-Informationen 59
4.20 MQP-Editor: Kontextmodell 60
4.21 MQP-Kontextmodell: Autor hinzufügen 61
4.22 MQP-Kontextmodell: Rolle hinzufügen 62
4.23 MQP-Kontextmodell: Artefakt hinzufügen 63
4.24 MQP-Neues Artefact: Neues Diagram hinzufügen 64
4.25 MQP-Neues Artefact: Neue Spezifikation hinzufügen 65
4.26 MQP-Neues Artefact: Neue Beziehung anlegen 65
4.27 MQP-Editor: ausgefülltes Kontextmodell 66
4.28 MQP-Editor: Informationsbedürfnissmodell 67
4.29 MQP-Informationsbedürfnissmodell: Neues Ziel anlegen 67
4.30 MQP-Editor: Qualitätsmodell 69
4.31 MQP-Qualitätsmodell: Palette der Zeichenfläche 69
4.32 MQP-Editor: Messmodell 71
4.33 MQP-Messmodell: Neue Basismessung eingeben 72
4.34 MQP-Messmodell: Neuen Indikator eingeben 74
4.35 MQP-Präsentationsmodell 75
4.36 Diagnose: GUI 77
4.37 Klassendiagrammeditor: Refactoringmenü 80

4.38 Klassendiagrammeditor: Fehlermeldung beim Aufruf des Refactoringmenüs 81
4.39 Refactoringassistent: Startseite 81
4.40 Refactoringassistent: Entscheidungsseite zur Angabe einer Zeichenfolge . . . 82
4.41 Refactoringassistent: Entscheidungsseite zur Auswahl von Elementen 82
4.42 Refactoringassistent: Entscheidungsseite zur Auswahl von Teilprädikaten . 83
4.43 Refactoringassistent: Erfolgreicher Abschluss eines Refactorings 83

5.1 Tutorial: Fertiges UML/Z-Modell 92
5.2 Tutorial: Projekt anlegen 93
5.3 Tutorial: Projektname 93
5.4 Tutorial: Diagrammtyp 94
5.5 Tutorial: Dateiname für das Diagramm 94
5.6 Tutorial: Elemente in der Outline 95
5.7 Tutorial: Eigenschaften des Datentyps Time 96
5.8 Tutorial: Eigenschaften der ZExpression Specification für Time 96
5.9 Tutorial: Z-Editor - ZExpression für Time 97
5.10 Tutorial: Eigenschaften des Datentyps Minute 98
5.11 Tutorial: Z-Editor - ZExpression für Minute 98
5.12 Tutorial: Constraint für Minute 99
5.13 Tutorial: Z-Editor - ZPredicate für Minute 99
5.14 Tutorial: Eigenschaften des Datentyps Hour 100
5.15 Tutorial: Eigenschaften des Datentyps Fachgebiet 100
5.16 Tutorial: Eigenschaften des Datentyps Text 101
5.17 Tutorial: Eigenschaften des Datentyps $\mathbb{N}$ 101
5.18 Tutorial: Eigenschaften des Datentyps Studiengang 102
5.19 Tutorial: Eigenschaften des Datentyps $\mathit{Text} \times \mathbb{N}$ 102
5.20 Tutorial: Die Zeichenfläche des Editors 103
5.21 Tutorial: Eigenschaften der Klasse Person 104
5.22 Tutorial: Eigenschaften des Attributes name 104
5.23 Tutorial: Eigenschaften des Attributes vorname 105
5.24 Tutorial: Eigenschaften des Attributes email 106
5.25 Tutorial: Eigenschaften der Klasse Student 106
5.26 Tutorial: Eigenschaften des Attributes Matrikelnummer 107
5.27 Tutorial: Eigenschaften des Attributes Studiengang 108
5.28 Tutorial: Eigenschaften der Methode lernen 108
5.29 Tutorial: Eigenschaften der Klasse Professor 109
5.30 Tutorial: Eigenschaften des Attributes semesterWochenStunden 110
5.31 Tutorial: Eigenschaften des Attributes gehalteneVorträge 110
5.32 Tutorial: Eigenschaften der Methode forschen 111
5.33 Tutorial: Eigenschaften der Methode vortragHalten 112
5.34 Tutorial: Eigenschaften der Methode mitarbeiterMotivieren 112
5.35 Tutorial: Eigenschaften der Methode prüft 113
5.36 Tutorial: Eigenschaften der Methode publiziertDokument 114
5.37 Tutorial: Eigenschaften der Methode hältZentralübung 114
5.38 Tutorial: Eigenschaften der Methode hältÜbung 115
5.39 Tutorial: Eigenschaften der Klasse Vorlesung 115
5.40 Tutorial: Eigenschaften des Attributes Zeit 116

5.41 Tutorial: Eigenschaften des Attributes Fachgebiet . . . 116
5.42 Tutorial: Eigenschaften des Attributes Name . . . 117
5.43 Tutorial: Eigenschaften des Attributes Raum . . . 118
5.44 Tutorial: Generalisierung Student . . . 118
5.45 Tutorial: Generalisierung Professor . . . 119
5.46 Tutorial: Assoziation hält . . . 120
5.47 Tutorial: Assoziation nimmt teil . . . 120
5.48 Tutorial: Neues allgemeines Projekt anlegen . . . 121
5.49 Tutorial: Neuen Modellqualitätsplan anlegen . . . 122
5.50 Tutorial: Modellqualitätsplan anlegen . . . 122
5.51 Tutorial: Modellqualitätsplan - grundlegende Einstellungen . . . 123
5.52 Tutorial: Kontextmodell anlegen . . . 124
5.53 Tutorial: Artefakt hinzufügen . . . 125
5.54 Tutorial: Informationsbedürfnismodell - Ziele . . . 126
5.55 Tutorial: Zuordnung Ziele zu Rollen . . . 127
5.56 Tutorial: Informationsbedürfnismodell . . . 127
5.57 Tutorial: Qualitätsmodell . . . 128
5.58 Tutorial: Qualitätsmodell - Zuordnung von Fragen . . . 129
5.59 Tutorial: Qualitätsmodell . . . 129
5.60 Tutorial: Zuordnung der Qualitätsattribute im Messmodell . . . 130
5.61 Tutorial: Eingabe einer neuen Messung . . . 130
5.62 Tutorial: Messmodell . . . 132
5.63 Tutorial: Auswahl des Modells . . . 132
5.64 Tutorial: Präsentationsmodell . . . 133
5.65 Tutorial: Benutzung der Diagnose . . . 134
5.66 Tutorial: Auswahl des Refactorings . . . 136
5.67 Tutorial: Startseite Wizard . . . 136
5.68 Tutorial: Seite mit dem Refactoring . . . 137
5.69 Tutorial: Auswahl der Klasse . . . 137
5.70 Tutorial: Methodenauswahl . . . 138
5.71 Tutorial: Selektion auszugliedernder Methoden . . . 138
5.72 Tutorial: Benennung neuer Klasse . . . 139
5.73 Tutorial: Beginn der Assoziation angeben . . . 139
5.74 Tutorial: Ende der Assoziation angeben . . . 139
5.75 Tutorial: Refactoring durchgeführt . . . 140

6.1 Komponentendiagramm: Architekturübersicht . . . 144
6.2 Klassendiagramm: IRefactoringEditor-Interface . . . 148
6.3 Klassendiagramm: IRefactoringDiagnostics-Interface . . . 149
6.4 Klassendiagramm: IOCL-Interface . . . 150
6.5 Klassendiagramm: IMQP-Interface . . . 151
6.6 Klassendiagramm: IZParser-Interface . . . 151
6.7 Paketdiagramm: UML/Z-Model . . . 154
6.8 Paketdiagramm: UML/Z-Model-Edit . . . 155
6.9 Paketdiagramm: UML/Z-Model-Editor . . . 156
6.10 Klassendiagramm: Realisierung von enable- und effect-Bedingungen . . . 156
6.11 Klassendiagramm: Realisierung der Delta-Liste in Operationen . . . 157

6.12 Klassendiagramm: Z-Konstruktor . . . 157
6.13 Klassendiagramm: Globale Z-Definitionen . . . 158
6.14 Klassendiagramm: Anbindung des Z-Parsers an das Modell . . . 159
6.15 Komponentendiagramm: GMF Editor . . . 160
6.16 Paketdiagramm: Klassendiagrammeditor . . . 161
6.17 Paketdiagramm: Zustandsdiagrammeditor . . . 161
6.18 Komponentendiagramm: MQP . . . 162
6.19 Paketdiagramm: Metamodell des Modellqualitätsplans . . . 163
6.20 Klassendiagramm: Meta-Kontextmodell . . . 164
6.21 Klassendiagramm: Meta-Informationsbedürfnismodell . . . 165
6.22 Klassendiagramm: Meta-Qualitätsmodell . . . 166
6.23 Klassendiagramm: Meta-Messmodell . . . 167
6.24 Klassendiagramm: Meta-Präsentationsmodell . . . 168
6.25 Komponentendiagramm: MQP-GUI . . . 169
6.26 Kompositionsstrukturdiagramm: MQP-Measurement . . . 170
6.27 Klassendiagramm: MQP-Measurement . . . 171
6.28 Klassendiagramm: Diagnostics-Komponente . . . 173
6.29 Klassendiagramm: Normalisierung . . . 174
6.30 Diagnose: Beispiel: lineare Normalisierung . . . 176
6.31 Diagnose: Beispiel: stückweise lineare Normalisierung . . . 176
6.32 Sequenzdiagramm: Verarbeitung der Indikatoren . . . 177
6.33 Sequenzdiagramm: Anzeige möglicher Normalisierungen . . . 178
6.34 Sequenzdiagramm: Berechnung von Diagnosen und passender Therapien . . . 179
6.35 Klassendiagramm: Refactoring-Komponente . . . 181
6.36 Klassendiagramm: von Condition abgeleitete Klassen . . . 183
6.37 Klassendiagramm: von CalculationTemplate abgeleitete Klassen . . . 184
6.38 Klassendiagramm: Decisions . . . 186
6.39 Klassendiagramm: Variablenmanagement in Refactorings . . . 187
6.40 Klassendiagramm: Klassen der Refactoring-GUI . . . 188
6.41 Klassendiagramm: Fortschrittsangaben bei Ausführung von Refactorings . . . 189
6.42 Sequenzdiagramm: Finden von Refactorings zu Modellelementen . . . 190
6.43 Sequenzdiagramm: Ausführen eines Refactorings . . . 191
6.44 Sequenzdiagramm: Ausführen eines komplexen Refactorings . . . 192
6.45 Klassendiagramm: OCLWrapper-Komponente . . . 193
6.46 Klassendiagramm: CZTWrapper-Komponente . . . 195

7.1 Installation: Plugins aus Liste Enabling Features . . . 199
7.2 Modell: Hinzufügen einer Operation . . . 206
7.3 Modell: Hinzufügen einer EAnnotation . . . 206
7.4 Modell: Source-Attribut der EAnnotation . . . 207
7.5 Modell: Hinzufügen eines Details Entry . . . 207
7.6 Modell: Attribute des Details Entry . . . 208
7.7 Modell: Generieren des UML/Z-Modell-Java-Codes . . . 210
7.8 Editor: GMF Generierung . . . 211
7.9 Editor: GMF-Mapping Modell . . . 212
7.10 Editor: GMF Genmodel . . . 213
7.11 Editor: GMF-Quelltext generieren . . . 213

7.12 Editor: Tab für GMF Genmodell 219
7.13 Editor: Tab für GMF Genmodell 219

9.1 Qualitätskreislauf 247

Tabellenverzeichnis

4.1 Z-Referenz: Prelude 48
4.2 Z-Referenz: Number Toolkit 50
4.3 Z-Referenz: Set Toolkit 51
4.4 Z-Referenz: Relation Toolkit 51
4.5 Z-Referenz: Function Toolkit 52
4.6 Z-Referenz: Sequence Toolkit 52
4.7 Refactoring: Nichtterminale der Grammatik 84

7.1 Entwicklung: Übersicht der Schnittstellen 202
7.2 Entwicklung: Übersicht der Komponenten 203
7.3 Refactoring: Implementierte Decisiontypen 240

Listings

4.1 Diagnoseknoten: <universalParameter> . . . 77
4.2 Diagnoseknoten: <symptom> . . . 78
4.3 Diagnoseknoten: <diagnosis> . . . 78
4.4 Diagnoseknoten: <therapy> . . . 78
4.5 Diagnoseknoten: <normalisation> . . . 79
5.1 Tutorial: OCL-Struktur für Messungen . . . 131
6.1 Diagnose: Beispiel für eine Schulnotennormalisation . . . 175
7.1 Editor: Init Statemachine auf Canvas . . . 215
7.2 Editor: Init Statemachine in Outline . . . 216
7.3 Editor: Parser für Transitionen: getString . . . 217
7.4 Editor: Parser für Transitionen: isAffectingEvent . . . 218
7.5 Editor: Z-Editor im RMC-Menü . . . 220
7.6 MQP: EditorPart . . . 224
7.7 MQP: Form-Konstruktor . . . 225
7.8 MQP: Section . . . 226
7.9 MQP: Lesezugriff auf das Modell . . . 228
7.10 MQP: CreateMQPAuthorCommand . . . 229
7.11 MQP: Verwendung CreateMQPAuthorCommand . . . 230
7.12 Diagnose: Beispiel: neues Symptom . . . 231
7.13 Diagnose: Beispiel: neue Diagnose . . . 232
7.14 Diagnose: Beispiel: neue Texttherapie . . . 232
7.15 Diagnose: Beispiel: neue Refactoringtherapie . . . 232
7.16 Diagnose: Beispiel: Struktur einer Normalisierung . . . 233
7.17 Diagnose: Anlegen einer neuen Normalisierung . . . 233
7.18 Grammatik-Abschnitt: Optionen von JavaCC . . . 235
7.19 Grammatik-Abschnitt: Definition der Klasse RefactoringParser . . . 235
7.20 Grammatik-Abschnitt: Whitespace . . . 236
7.21 Grammatik-Abschnitt: Tokenliste . . . 236
7.22 Grammatik-Abschnitt: interne Tokenliste . . . 237
7.23 Grammatik-Abschnitt: interne Tokenliste . . . 238
7.24 Grammatik-Abschnitt: globale OCL-Statements . . . 239
7.25 Grammatik-Abschnitt: Query . . . 239
7.26 Grammatik-Abschnitt: Parameter der Query . . . 239
7.27 Grammatik: Liste der Typen von Decisions . . . 241

Teil I

Konzeptionelle Sicht

1 Einleitung

„Wie. . . ? Du liest das alles?“

Hendrik, zu Thomas

1.1 Motivation

Modelle werden heute in vielfältiger Weise genutzt, um beispielsweise Prozesse aus dem realen Leben auf einer abstrakten Ebene zu entwerfen und zu untersuchen. Insbesondere gilt dies für Softwaremodelle, deren Bedeutung stetig zunimmt. Aus diesem Grund steigen die Anforderungen an die Softwaremodelle immer weiter, und die erstellten Modelle müssen bestimmte Kriterien erfüllen. Ein wichtiger Aspekt ist hierbei die Qualität des vorhandenen Modells.

Die Ziele der Projektgruppe ModelCockpit waren der Entwurf und die Implementierung eines adäquaten Messinstruments, mit dem ein eigenes Verständnis von Qualität modelliert werden kann. Dieses *Qualitätsmodell* sollte dann als Basis für eine Untersuchung eines gegebenem Softwaremodells dienen. Verschiedene Qualitätsmodelle sind deshalb notwendig, weil verschiedene Rollen eine unterschiedliche Ansicht von Qualität haben und dabei auf verschiedene Aspekte Wert legen. Ein weiterer Punkt, der bei Qualitätsmodellen zu beachten ist, ist der, dass je nach Anwendungsbereich unterschiedliche Anforderungen an ein Qualitätsmodell gestellt werden. So verfügt eine Anwendung im Hochsicherheitsbereich über ganz andere Standards als eine Anwendung, die an einer Hochschule eingesetzt wird. Daher muss die Aufarbeitung der Messergebnisse so ausgelegt sein, dass jeder Adressat die für ihn vorgesehenen Messergebnisse verwenden kann.

Die Ziele der Projektgruppe RefaSo waren der Entwurf und die Implementierung eines Werkzeugs, mit dem Änderungen an einem gegebenen Modell vorgenommen werden können, welches in einer formalen Sprache spezifiziert ist. Die Änderung eines Modells respektive Programms wird auch als Refactoring bezeichnet. Refactorings verbessern durch eine *verhaltenserhaltende Änderung* der vorhandenen Struktur die Lesbarkeit, Wartbarkeit und Adaptierbarkeit. Genauer beschreibt es Martin Fowler in seinem Buch „Refactoring: Improving the Design of Existing Code“[1]:

[1] Fowler bezieht sich in seinem Buch auf Quelltext. Das Verständnis von Refactorings kann jedoch auch auf Softwaremodelle übertragen werden.

> „Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the changes of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.“
>
> [Fow99]

Auf Basis einer Analyse, welche mittels eines Modellqualitätsplans vorgenommen wird, können Mängel am vorhandenen Modell aufgedeckt werden. Die Identifizierung von Modellmängeln bietet Ansatzpunkte für Änderungen, die als Verbesserungen des Modells vorgeschlagen werden können. Derartige Verbesserungsvorschläge können demnach auch *verhaltenserhaltende* Modelländerungen und damit Refactorings sein.

Die Qualität und Änderbarkeit eines Modells besitzen Verknüpfungspunkte, die durch die Zusammenlegung der Projektgruppen ModelCockpit und RefaSo herausgearbeitet werden sollten. Daher kommt auch die Bezeichnung RMC des entwickelten Tools, ein Kürzel stellvertretend für die beiden Projektgruppen Refaso und ModelCockpit.

1.2 Inhaltliche Übersicht

Diese Abschlussdokumentation ist in drei große Kapitel aufgeteilt, die jeweils einer Sichtweise entsprechen. Die Sichten ergeben sich wie folgt:

- konzeptionelle Sicht (S. 21)
- anwenderorientierte Sicht (S. 31)
- entwicklerorientierte Sicht (S. 141)

Die konzeptionelle Sicht wird in den Kapiteln Ideen und Konzepte (S. 25) vorgestellt. Dort werden die wichtigsten Ideen und Vorteile von RMC präsentiert.

Für den Anwender sind besonders die Kapitel Benutzerhandbuch (S. 35) und Tutorial (S. 91) wichtig. Neben einer Installationsanleitung erhält der Leser auch Kenntnis über die praktische Anwendung des entwickelten Tools. So werden die grundlegenden Funktionen im Benutzerhandbuch erläutert, und im Tutorial wird der Leser durch ein durchgehendes Anwendungsbeispiel geleitet.

Die letzten Kapitel beschäftigen sich mit der Architektur (S. 143) des entwickelten Tools und enthalten eine Dokumentation für Entwickler (S. 243) sowie eine API-Dokumentation (S. 243). Diese sind besonders für Entwickler interessant und auch bewusst für jene konzipiert worden. Hier sind Einstiegspunkte zu finden, um die Weiterentwicklung des Tools voranzutreiben.

2 Konzepte

„Nichts ist immer gleich true.“

Thomas

2.1 Die Sprache UML/Z

Die Realisierung von RMC hat gewisse Anforderungen, welche im Kapitel 1.1 beschrieben wurden. Um sie zusammenzuführen, musste eine geeignete Sprache zur Softwaremodellierung gefunden werden. Da keine bekannte Sprache alle Anforderungen erfüllen konnte, wurden Teile von bestehenden Sprachen zu *UML/Z* kombiniert. Die genaue Zusammensetzung erklärt der nächste Abschnitt.

UML/Z ist eine Abwandlung der Modellierungssprache UML[1]. Die Elemente von UML/Z sind derart verändert, dass sie für diese beiden Sichten CSP-OZ als *Semantik* verwenden können. CSP-OZ ist eine Kombination der Prozessalgebra CSP[2] und der formalen Sprache Object-Z (OZ). Dieses wiederum ist aus Z hervorgegangen, indem Z um Objektorientierung ergänzt wurde. Eine grundlegende Einführung in Z wird in [WD96] gegeben. Hierbei wird CSP genutzt, um Klassendiagramme zu beschrieben, Object-Z hingegen beschreibt die Zustandsdiagramme.

Daraus ergeben sich folgende Änderungen von UML/Z gegenüber der UML:

OZ-Typen: In UML/Z werden statt UML-Typen nur in Z definierte Typen verwendet. Dies gewährleistet, dass durch die mathematischen Definitionen der Typen präzise Sachverhalte darstellbar sind.

Pre- und Postconditions: Mittels dieser Conditions lassen sich Einschränkungen für das Modell spezifizieren. Sie werden mittels Z-Notation beschrieben.

Konstruktoren: Es muss genau ein Konstruktor pro Klasse existieren, damit die initialen Werte der Attribute festgelegt werden können.

Klasseninvarianten: In UML/Z gibt es analog zu UML auch Klasseninvarianten. Diese Klasseninvarianten werden mittels OZ-Ausdrücken beschrieben.

[1] UML wird spezifiziert von der Object Management Group (OMG).
[2] Communicating Sequential Processes

Parameter: Neben den Parameterarten **input** und **output**, die auch in UML definiert sind, gibt es den Parameter **simple**. Dieser Parameter nimmt die Variablen auf, die innerhalb einer Methode genutzt werden und die keine **input**- und **output**-Parameter sind.

Delta-Listen: Die Delta-Listen geben für jede Methode an, welche Klassenattribute von dieser Methode geändert werden.

Globale Z-Definitionen: Um Redundanzen in den Z-Ausdrücken zu verhindern, lassen sich globale Definitionen in Z definieren und nutzen.

Transitionen: Einer Transition darf lediglich eine Methode zugeordnet werden. Guards und Trigger, wie aus UML bekannt, sind in UML/Z nicht definiert.

Details über die Realisieung dieser Eigenschaften von UML/Z finden sich in Kapitel 6.3.1.1 auf Seite 155.

Die Semantik von UML/Z wird in CSP-OZ definiert. Aufgrund dieser Eigenschaften können in CSP-OZ bewiesene Refactorings auch in UML/Z definiert und ausgeführt werden. Die Modelle selbst können jeweils von der einen in die andere Sprache übertragen werden.

Mittels UML/Z beschreibt der Anwender von RMC sowohl die Struktur als auch das Verhalten eines Modells. Ähnlich wie UML bietet auch UML/Z dafür verschiedene *Sichten* auf das Modell, die jeweils nur einen Ausschnitt liefern. Angelehnt an UML gibt es zwei Sichten:

- Die Sicht **Klassendiagramm** zeigt die Struktur, die aus Klassen besteht, die mittels Assoziationen verbunden sind. Klassen enthalten Methoden und Attribute.
- Ein **Zustandsdiagramm** zeigt hingegen den Ablauf *genau einer* Klasse. Die Transitionen spiegeln Methodenaufrufe wieder und verbinden einzelne Zustände untereinander. Superstates und andere bekannte Konzepte sind auch möglich.

Weiterführende Literatur zum Themenkomplex Refactorings findet sich unter [Weh00] von Prof. Dr. Heike Wehrheim. Informationen über Modellqualitätspläne bieten die Arbeiten von Hendrik Voigt [VE08].

2.2 Qualitätskreislauf

Ein Softwaremodell soll vor allen Dingen eine Eigenschaft erfüllen: es muss den Qualitätsansprüchen genügen. Um diese Ansprüche zu erfüllen, muss man die aktuelle Qualität kennen, um im nächsten Schritt die *gewünschte* Qualität zu erreichen. Wie man dieses Ziel als Prozess ausdrücken kann, zeigt dieses Kapitel.

Ein typisches Szenario für die Verbesserung von Modellqualität besteht aus folgendem Ablauf. Die Qualität muss zunächst definiert und modelliert werden. Mit diesem Qualitätsverständnis wird eine Messung und Beurteilung der Qualität des Modelles durchgeführt. Die erhaltenen Ergebnisse müssen dem Anwender präsentiert und die gefundenen Mängel sollten behoben werden. Das resultierende verbesserte Modell kann als Ausgang für eine neue Messung der Qualität dienen. Der Prozess beginnt erneut. Dieses Vorgehen ist als Qualitätskreislauf bekannt.

Abbildung 2.1 zeigt den Kreislauf, den man mit RMC vollziehen kann. Dieser Prozess lässt sich mit RMC bewerkstelligen, ohne auf externe Programme zurückgreifen zu müssen.

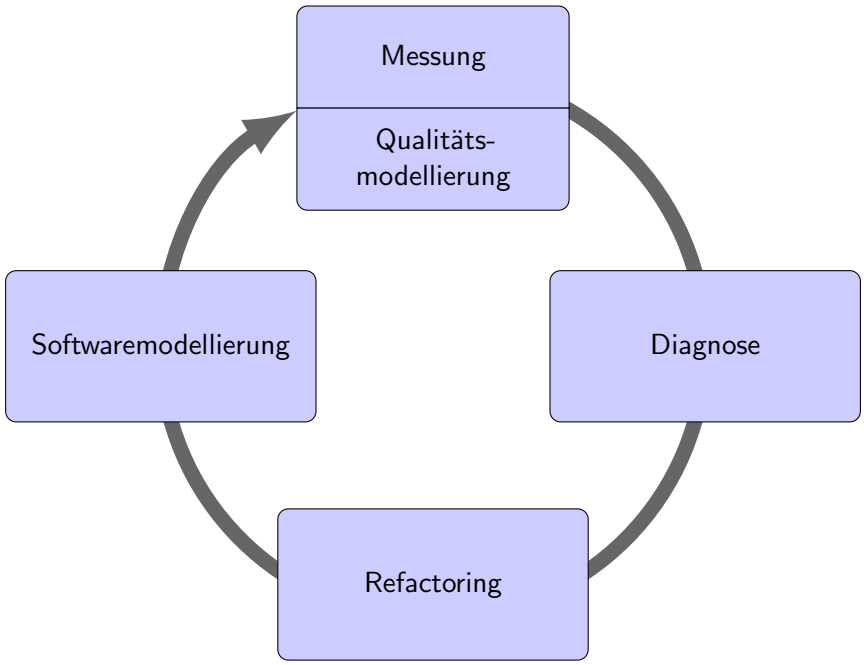
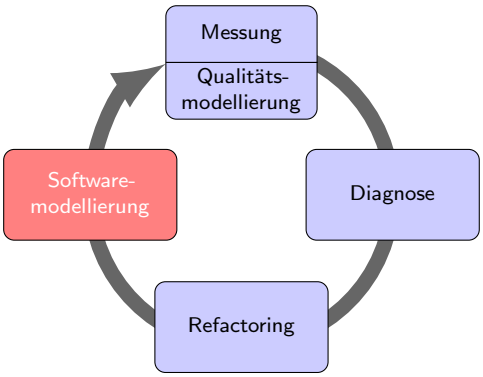


Abbildung 2.1: Qualitätskreislauf

Dieser Ablauf ist nicht zwingend und soll nur eine Möglichkeit aufzeigen, was RMC zu leisten vermag. Andere Abläufe inklusive andersartiger Iterationen sind somit ohne Probleme möglich und auch gewollt. Der gezeigte Kreislauf wird nicht explizit von RMC unterstützt; es existieren also keine Sichten, die die einzelnen Schritte von diesem Ablauf wiedergeben.

Es gibt zwei verschiedene Datenhierarchien, auf denen RMC arbeitet, die im Folgenden beschrieben werden.



UML/Z-Modell

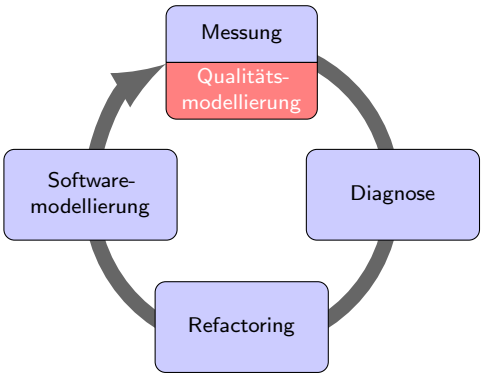
Das UML/Z-Modell beinhaltet die Daten, auf denen Verbesserungen und Messungen durchgeführt werden sollen. Dieses Modell wird innerhalb von RMC mittels des Modelleditors erstellt. Als Ausgangspunkt für einen Durchlauf des Qualitätskreislaufs muss ein solches Modell existieren.

Für die beiden Sichten von UML/Z, Klassen- und Zustandsdiagramme, gibt es jeweils einen Editor.

Modellqualität

Die Modellqualität wird im Modellqualitätsplan erstellt und bearbeitet. Dies geschieht unabhängig vom Modell, *auf dem* gemessen werden soll, und ist daher losgelöst vom eigentlichen Modellinhalt des UML/Z-Modells.

2.2.1 Modellieren der Qualität



Um die Qualität eines konkreten Softwaremodells zu ermitteln, muss man zuvor die Vorstellungen von Qualität modellieren. Dieser Prozess wird im Folgenden beschrieben, indem auf die fünf Teilmodelle der Modellqualität eingegangen wird.

Für einen Durchlauf des Qualitätskreislaufs muss zunächst eine Qualitätsdefinition geschaffen werden, indem man die Qualität modelliert.

Dabei existieren folgende Probleme:

- Es gibt kein allgemeingültiges Rahmenwerk für Modellqualität.
- Unterschiedliche Anwendungsfälle erfordern unterschiedliche Rahmenwerke.
- Bereits definierte Qualitätsvorstellungen müssen für neue Anwendungen angepasst werden.

Diese Anforderungen an Qualitätsdefinitionen zeigen deutlich die Nachteile für eine statische Beschreibung von Qualität auf. Eine flexible Modellierung bietet jedoch einen guten Lösungsansatz. Modellqualitätspläne (MQP) bieten eine solche Lösung, um Modellqualität zu beschreiben, zu messen und aufzubereiten.

Wie ist nun ein Modellqualitätsplan aufgebaut? Ein MQP besteht aus 5 Teilmodellen, die den Prozess der Modellierung und Messung von Qualität widerspiegeln:

Kontextmodell: In welchem Kontext soll die Qualität bestimmt werden? Ein Modell kann in unterschiedlichen Modellierungssprachen verfasst sein, unterschiedliche Rollen sind an der Erstellung beteiligt. Ebenso ist der Anwendungszweck des Modelles zu berücksichtigen. Das Kontextmodell realisiert die Modellierung dieser Parameter.

Informationsbedürfnismodell: Welches Interesse an Informationen über die Qualität besteht? Verschiedene Rollen haben nicht die gleichen Anforderungen an Qualität. Unterschiedliche Interessen benötigen jedoch auch unterschiedliche Messungen der Qualität. Es ist also notwendig, Anforderungen an die Qualität zu modellieren. Dies wird im Informationsbedürfnismodel dargestellt.

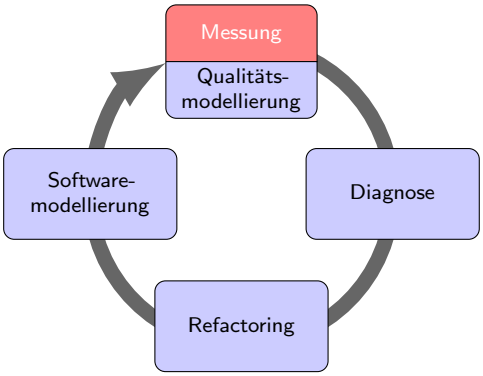
Qualitätsmodell: Was soll gemessen werden? Modelle haben Eigenschaften. Welche dieser Attribute gemessen werden sollen, um Qualitäten des Modelles zu bestimmen, wird durch das Qualitätsmodell definiert.

Messmodell: Wie wird gemessen? Das Messmodell repräsentiert die Messungen, die auf dem Modell durchgeführt werden. Die Messungen sind ausführlicher in Kapitel 2.2.2 beschrieben.

Präsentationsmodell: Wie werden die Messergebnisse dargestellt? Die Modellierung von Qualität und das Ausführen von Messungen liefert Ergebnisse und Aussagen über die Modellqualität. Die in den vorherigen Modellen definierten Bedürfnisse an Qualität und die Kontexte der Messungen führen zu unterschiedlichen Anforderungen an die Darstellung und Auswertung der Modellqualität. Diesem Anspruch soll das Präsentationsmodell Rechnung tragen.

Mit einem so definierten Modellqualitätsplan kann nun der nächste Schritt im Qualitätskreislauf durchgeführt werden: die Messung der Qualität.

2.2.2 Messen der Qualität



Nach der Modellierung der Qualität lässt sich die Qualität eines Softwaremodells feststellen. Dies wird durch das Ausführen von *Messungen* realisiert, welche im Qualitätsmodell definiert wurden.

Um die Qualität des Modells zu erhalten, werden *Messungen* durchgeführt. Ein Durchlauf von Messungen besteht aus vielen Teilmessungen, die teilweise aufeinander aufbauen.

Die duchzuführenden Messungen können unabhängig von dem Modell erstellt und bearbeitet werden. Der Anwender kann daher die Arbeiten am Modell und das Modellieren der Qualität parallel durchführen.

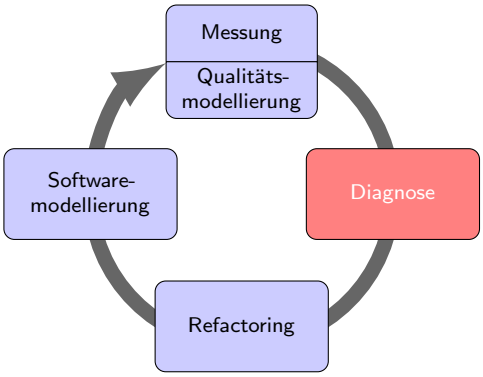
Die Messungen werden durch folgende verschiedene *Messungstypen* abgebildet:

Basismessungen sind die grundlegenden Bausteine der Messungen, die keine anderen Messungen nutzen.

Abgeleitete Messungen werden aus Basismessungen und anderen abgeleiteten Messungen zusammengesetzt.

Indikatoren bewerten eine oder mehrere Messungen anhand von Regeln, die der Benutzer definiert hat. Sie geben eine Interpretation des Messergebnisses zurück.

2.2.3 Verbessern der Qualität

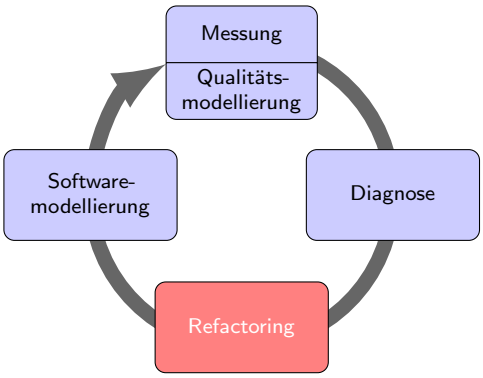


Die Ergebnisse der Messung werden nun zur Verbesserung des Softwaremodells verwendet, da aus den Messungen mögliche Mängel des Modells festgestellt werden können. Diese Mängel lassen sich dann beseitigen.

Die Ergebnisse von Messungen, wie sie in Kapitel 2.2.2 auf Seite 29 beschrieben wurden, eignen sich dazu, das gemessene Modell zu verbessern, da bei einer Messung auch Mängel des Modells aufgedeckt werden. Diese Mängel möchte man in der Regel beseitigen, um das Modell zu verbessern. Die Bedingung dafür ist, dass der Nutzer über die Mängel in einer Form informiert wird, so dass er die Mängel leicht nachvollziehen und daraus Schlüsse ziehen kann. Dies ist die Zielsetzung der Diagnose.

Aufgabe dieser Komponente ist es, aus den einzelnen Messungen auf konkrete Mängel zu schließen und Verbesserungen anzubieten. Die Mängel werden also dem Nutzer präsentiert; sie basieren auf dem Stand des Modells zum Zeitpunkt der Messung und schenken daher den nachträglichen Änderungen keine Beachtung.

Es gibt verschiedene Arten von Verbesserungen, die von der Diagnose angeboten werden.



Eine Art von Vorschlag ist ein *Refactoring* - dies bezeichnet die Änderung eines Modells, bei der die nach außen sichtbare Struktur und das Verhalten des Modells gleich bleiben. Ein Refactoring stellt also eine Art Umstrukturierung des Modells dar.

Der Anwender kann frei entscheiden, ob er das Modell mittels der angebotenen Vorschläge ändert oder eigenständige Verbesserungen durchführt. Dadurch kann jeder Nutzer gemäß seiner bevorzugten Arbeitsweise handeln und wird vom Programm nicht eingeschränkt.

An diesem Punkt im Kreislauf kann der Nutzer beliebig viele der angebotenen Vorschläge ausführen, ohne nochmals zum Messen der Qualität oder der Diagnose zurückkehren zu müssen.

Teil II

Anwenderorientierte Sicht

3 Installationsanleitung

„I will start cooking coffee, in case I find the Thermoskanne.“

Christian, zu Ping

Damit das Tool RMC verwendet werden kann, stellt die Projektgruppe RMC dem Benutzer für die beiden Betriebssysteme Microsoft Windows und GNU/Linux auf ihrer Internetseite im Downloadbereich[1] jeweils ein Paket für das entsprechende Betriebssystem zum Download bereit. Nach der Auswahl und dem Herunterladen des entsprechenden Installationspakets, muss das Paket nur noch mit einem Packprogramm wie beispielsweise 7zip oder unzip entpackt werden.

Nachdem Entpacken des Installationspakets, ist die Datei `rmc.exe` (unter Windows) bzw. `rmc` (unter Linux) im erstellten Verzeichnis zu finden. Durch Klicken auf diese Dateien startet das Tool RMC.

> **Hinweis:** Erforderlich für das Nutzen des Tools RMC ist ein Java Runtime Environment (JRE) oder Java Development Kit (JDK). Beide Pakete sind auf den Downloadseiten von SUN kostenlos erhältlich.

[1] http://www.cs.uni-paderborn.de/fachgebiete/fg-engels/lehre/ws0708/pg-rmc-teil2/download.html

4 Benutzerdokumentation

„Deine Freundin kann doch mithelfen... Kann die Tippen?“

Bernhard, zu Martin

4.1 Einleitung

Dieses Kapitel soll dem Benutzer des RMC-Tools als Dokumentation dienen. Dabei werden dem Benutzer Informationen an die Hand gegeben, die notwendig sind, um das RMC-Tool nutzen zu können. Anschließend werden viele hier erwähnte bzw. beschriebene Bereiche in einem Tutorial (Kapitel 5) noch einmal aufgegriffen und praktisch angewendet.

4.2 Komponenten

Für die Benutzung des RMC-Tools wird immer ein Projekt benötigt. Hat der Benutzer noch kein Projekt angelegt, so wird in Abschnitt 4.2.1 darauf kurz näher eingegangen. Abschnitt 4.2.2 beschreibt die einzelnen Elemente der ersten großen Komponente, dem Editor. Hier wird zuerst darauf Bezug genommen, wie ein neues Diagramm einem Projekt zugeordnet bzw. hinzugefügt wird, und was beim Anlegen zu beachten ist. Hat man ein Projekt und die gewünschten Diagramme angelegt, werden in Abschnitt 4.2.2.2 und 4.2.2.3 die einzelnen Elemente der Editoren (Klassen- und Zustandseditor) und deren Bedienung beschrieben. In Abschnitt 4.2.3 wird die zweite große Komponente des RMC-Tools näher beschrieben. Dieser Abschnitt befasst sich mit den einzelnen Modellen (Abschnitt 4.2.3.1 bis 4.2.3.7) eines Modellqualitätsplanes und den darin enthaltenen Bereichen. Mit Hilfe von Messergebnissen aus dem Modellqualitätsplan, kann dann die Diagnose aus dem Präsentationsmodell (siehe Kapitel 4.2.3.7) gestartet werden, welche in Abschnitt 4.2.4.1 beschrieben wird. Als letzte große Komponente wird in Abschnitt 4.2.5 auf die Refactoringkomponente näher eingegangen. Der letzte Abschnitt (Abschnitt 4.3) befasst sich in Form einer FAQ dann noch mit bekannten Problemen und dazu gehörigen Lösungen, die während der Nutzung des Tools auftreten können.

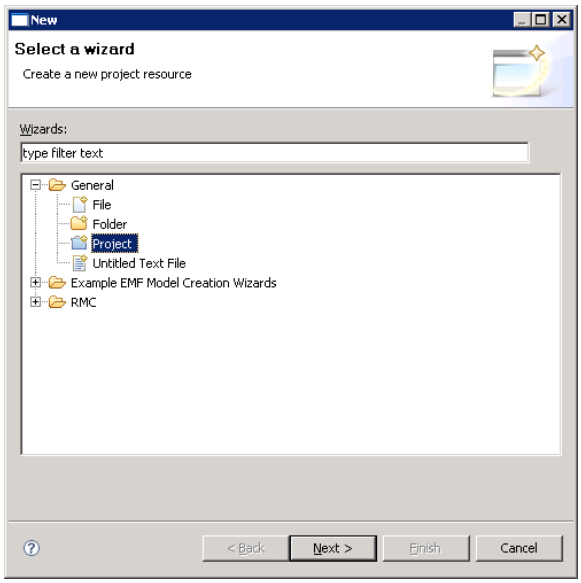


Abbildung 4.1: Projekt-Wizard: Startseite um ein neues Projekt anzulegen

4.2.1 Projekte

Es wird immer ein Projekt benötigt. Es kann ein schon vorhandenes Projekt genutzt oder ein neues angelegt werden. Dazu wählt der Anwender ⟨`File`⟩→⟨`New`⟩→⟨`Project`⟩ im Hauptmenü aus. Es erscheint ein Dialog in dem **Project** selektiert und mit [Next] bestätigt wird. Dies ist in Abbildung 4.1 dargestellt.

Auf der nun folgenden Dialogseite (Abbildung 4.2) wird der Name des Projektes eingegeben. Optional ist eine Änderung des Pfades möglich. Mit [Finish] wird das Erstellen des Projektes beendet.

4.2.2 Editor

Der grafische Editor ermöglicht das Anlegen von UML/Z-Modellen[1]. Ein Diagramm besteht immer aus zwei Dateien:

- dem eigentlichen Modell (`*.umlz`)
- der grafischen Notation (`*.umlz_state_diagram` oder `*.umlz_class_diagram`)

4.2.2.1 Diagramme anlegen

Es gibt zwei Wege ein Diagramm zu erstellen. Der Benutzer kann entweder ein neues Modell anlegen oder alternativ ein bereits vorhandenes Modell nutzen. Das Anlegen eines Diagramms inklusive eines neuen Modells stellt die übliche Vorgehensweise dar.

[1] Informationen über das Konzepept von UML/Z sind im Kapitel 2.1 verfügbar, Details über die Architektur finden sich in Kapitel 6.3.1

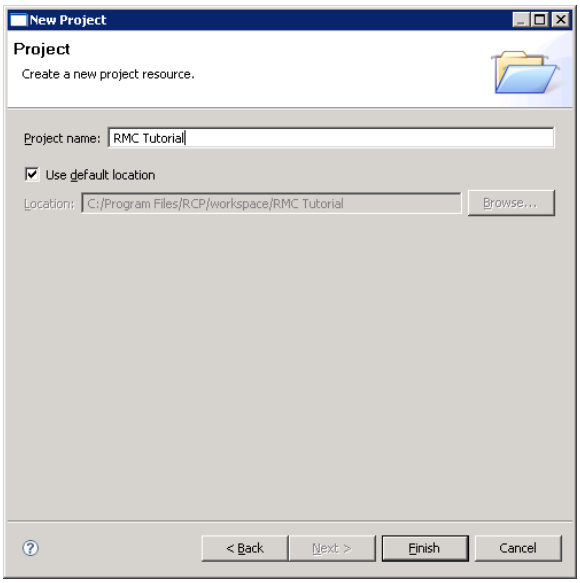


Abbildung 4.2: Projekt-Wizard: Projektname und Pfad festlegen

Diagramm samt neuem Modell erzeugen

Um ein neues Diagramm anzulegen, wird ein **Project** benötigt. Es kann ein vorhandenes genutzt oder ein neues angelegt werden. Das Anlegen eines Projektes ist in Kapitel 4.2.1 beschrieben. Alternativ kann ein bereits existierendes Projekt genutzt werden.

Dieses Projekt dient nun für die Aufnahme des neuen Diagramms. Dazu wählt der Anwender im Hauptmenü ⟨`File`⟩→⟨`New`⟩→⟨`Other`⟩ und wählt dort die Kategorie **RMC**. Um ein neues Klassendiagramm anzulegen, wird der Eintrag **UMLZ Classdiagram** selektiert, für ein Zustandsdiagramm wird **UMLZ Statediagram** gewählt. Abbildung 4.3 stellt diesen Dialog dar. Mit [Next] geht es weiter.

Abbildung 4.4 zeigt den Dialog für den Diagrammnamen und das Projekt, in welchem das neue Diagramm gespeichert werden soll.

Der Editor speichert die Inhalte des Modells in einem zweiten Modell, getrennt von den grafischen Inhalten. Im folgenden Schritt wird dieses Modell für das erstellte Diagramm angelegt, wie in Abbildung 4.5 veranschaulicht.

Mit [Finish] startet die Erzeugung der Dateien für das Modell und das Diagramm. Es öffnet sich der Editor mit dem gerade angelegten Diagramm.

Diagramm auf einem bestehenden Modell erzeugen

Wenn ein UML/Z-Modell bereits vorhanden ist, können auf diesem auch Diagramme erzeugt werden. Abbildung 4.6 zeigt, wie im **Project Explorer** durch einen Klick mit der rechten Maustaste das Kontextmenü geöffnet wurde.

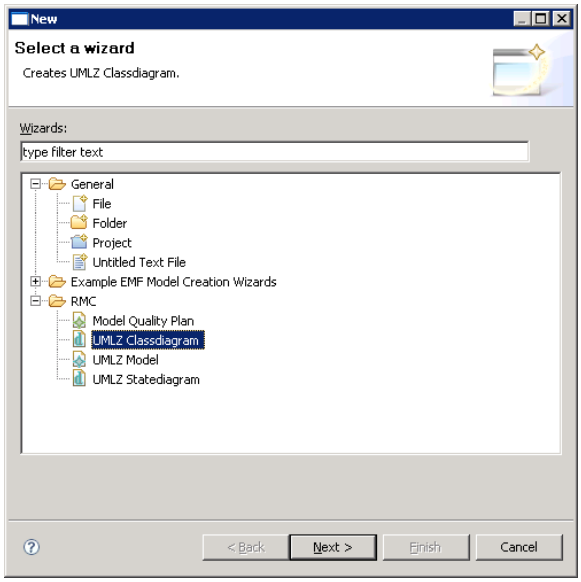


Abbildung 4.3: Projekt-Wizard: Diagrammtyp auswählen

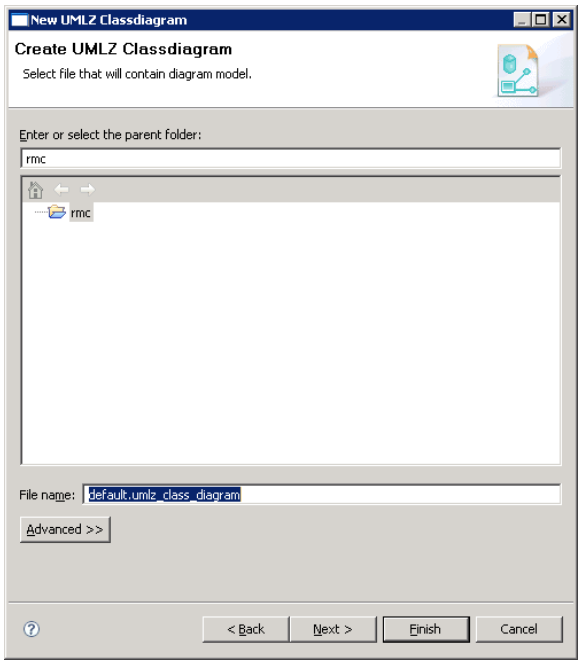


Abbildung 4.4: Projekt-Wizard: Name und Pfad des Diagrammes festlegen

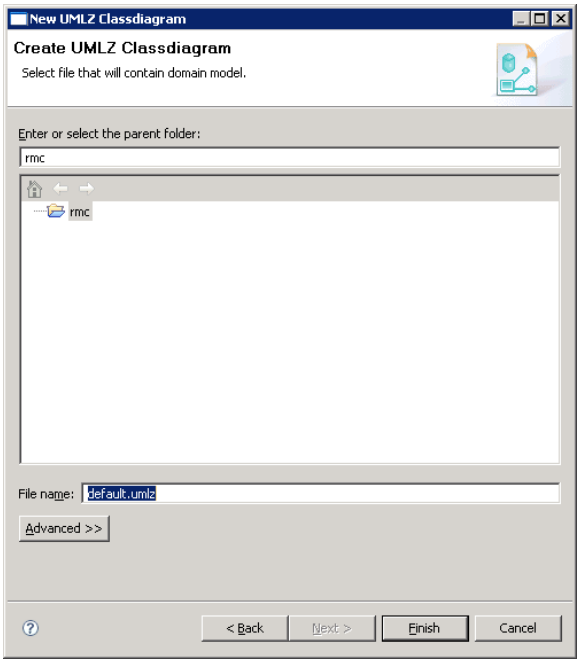


Abbildung 4.5: Projekt-Wizard: Modell für das neue Diagramm bestimmen

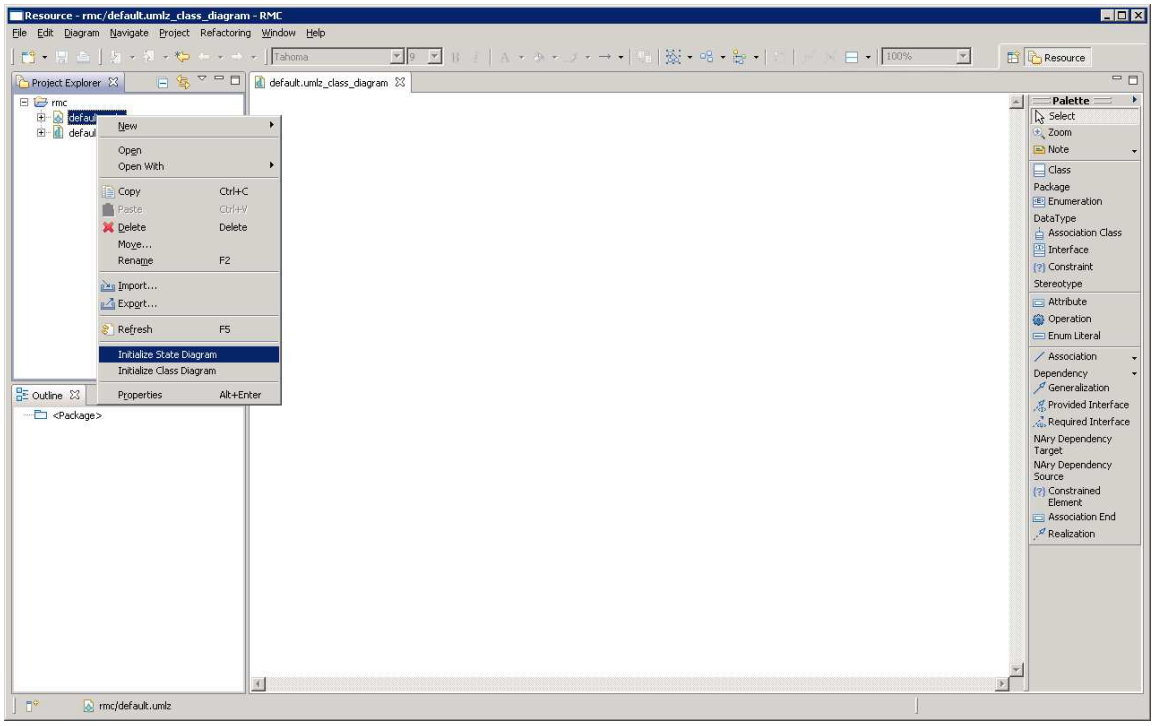


Abbildung 4.6: Anlegen eines Diagrammes auf einem bestehenden Modell

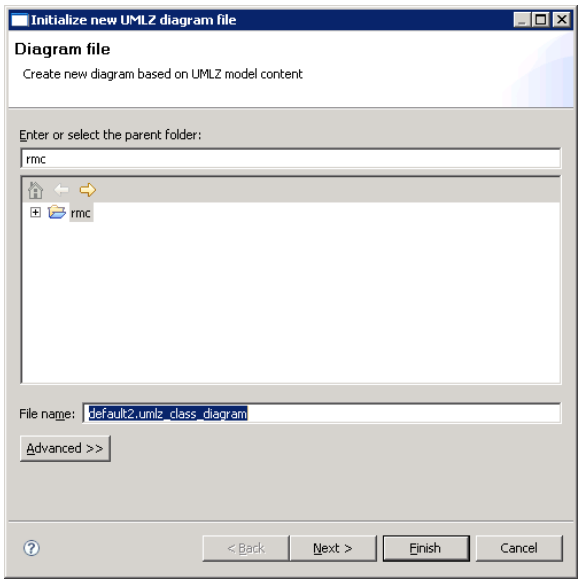


Abbildung 4.7: Projekt-Wizard: Name und Pfad für das Diagramm festlegen

Der Benutzer wählt nun ⟨`Initialize Class Diagram`⟩ oder ⟨`Initialize State Diagram`⟩, um ein Diagramm zu erstellen. Es erscheint ein Dialog (Abbildung 4.7), in dem der Dateiname für das neue Diagramm festgelegt wird.

Mit [Next] wird der nächste Dialog geladen. In diesem wählt der Benutzer das Element im Modell aus, welches als oberstes grafisches Element die Zeichenfläche darstellen soll.

Abbildung 4.8 stellt diese Elementauswahl dar. Zulässige Auswahlen sind:

- Klassendiagramm: **Package**
- Zustandsdiagramm: **Class**

> **Hinweis:** Mit ⟨`Window`⟩→⟨`Reset Perspective`⟩ können der Zustand und die Position der Fenster in ihren ursprünglichen Zustand versetzt werden.

4.2.2.2 Klassendiagramme

Erster unterstützter Diagrammtyp ist das Klassendiagramm. Abbildung 4.9 zeigt den UML/Z-Editor mit einem geöffneten Klassendiagramm.

Der Editor umfasst folgende Bereiche:

Canvas: Der Canvas ist die Zeichenfläche, auf dem die Elemente des Diagrammes als Grafiken erscheinen.

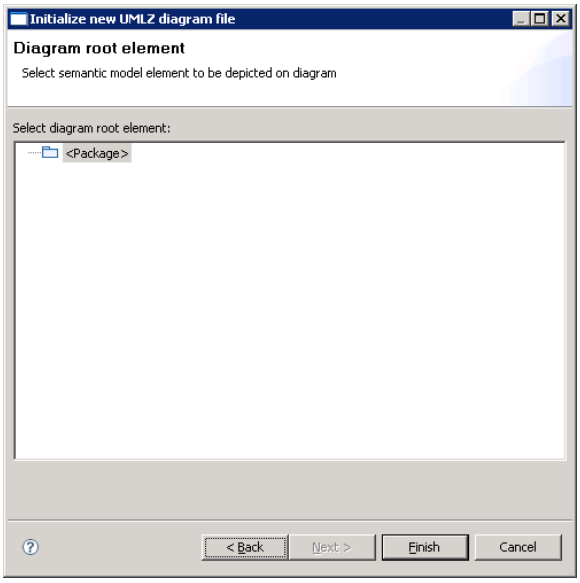


Abbildung 4.8: Projekt-Wizard: Wurzelelement des Diagrammes auswählen

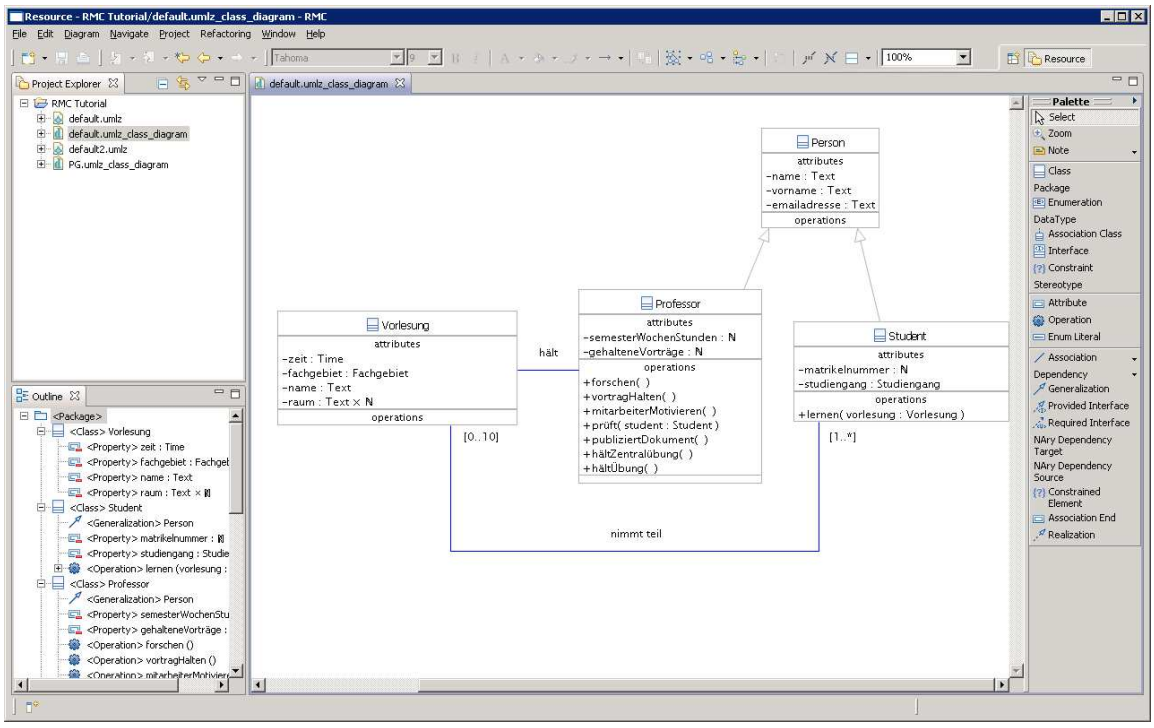


Abbildung 4.9: Übersicht über einen Klassendiagrammeditor

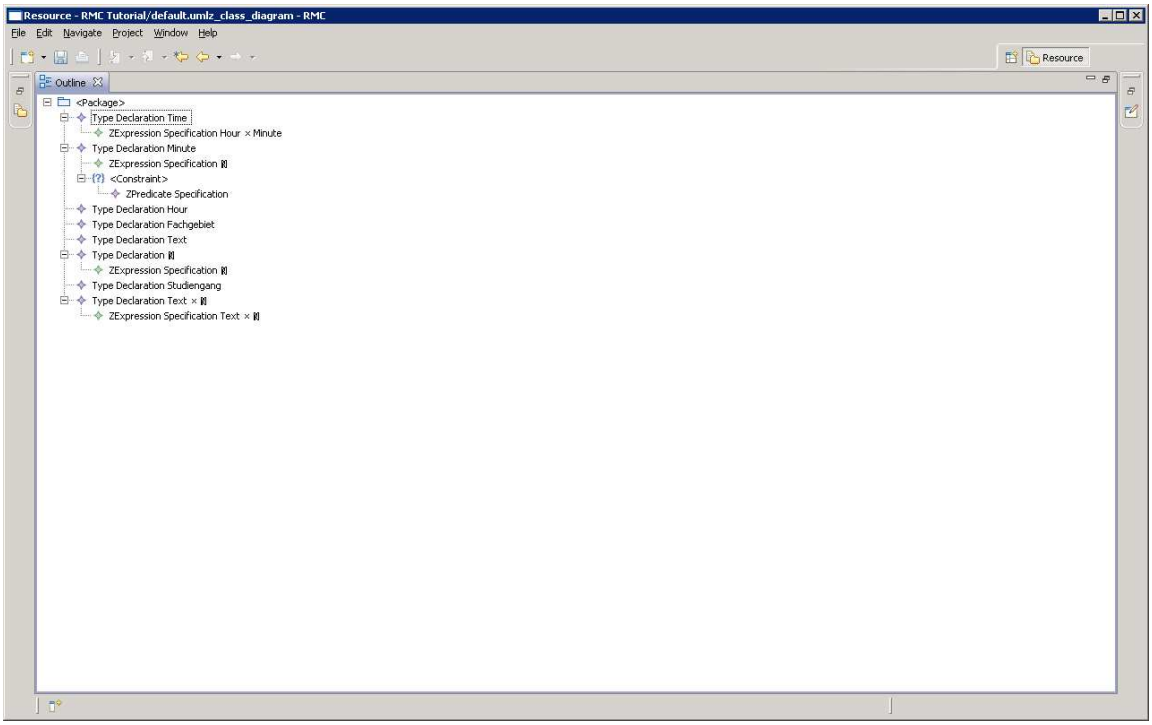


Abbildung 4.10: Die Outline des Klassendiagrammeditors

Palette: Die Palette am rechten Rand des Canvas ist das Werkzeug, um Elemente anzulegen.

Properties: In den Properties werden Informationen über das Element dargestellt, das gerade selektiert ist. Wenn die **Propertyview** nicht vorhanden ist: ⟨`Windows`⟩ → ⟨`Show View`⟩ →⟨`Properties`⟩.

Outline: In der Outline stehen die Kindelemente des gerade in der Zeichenfläche selektierten Elementes. Die Outline, dargestellt in Abbildung 4.10 ist ein zentrales Element für das Bearbeiten von Elementen, die nicht direkt auf dem Canvas sichtbar sind. Wenn die Outlineview nicht sichtbar ist: ⟨`Window`⟩→⟨`Show View`⟩→⟨`Outline`⟩

Project Explorer: Der Project Explorer bietet einen Überblick über die Projekte. Die einzelnen Modelle und Diagramme sind aufgelistet.

> **Hinweis:** Falls ein Fenster (eine **View**) fehlen sollte: Unter ⟨`Window`⟩→ ⟨`Show View`⟩ können Fenster neu geöffnet werden.

Anlegen von Elementen über die Palette

Die Elemente eines Klassendiagramms können über die Palette angelegt werden. Dazu selektiert der Anwender das gewünschte Element in der Palette und klickt auf den Canvas.

Elemente die zwei Knoten verbinden, werden durch Ziehen einer Verbindung zwischen Elementen auf dem Canvas erstellt. Folgende Symbole finden sich in der Palette:

Class
Legt eine neue Klasse an. Sobald diese auf dem Canvas erscheint, ist das Feld für den Klassennamen selektiert, und der Name kann eingegeben werden.

Enumeration
Legt einen neuen Aufzählungstyp an. Sobald dieser auf dem Canvas erscheint, ist das Feld für den Namen selektiert, und dieser kann eingegeben werden.

Association Class
Legt eine neue Assoziationsklasse an. Der Klassenname ist selektiert und kann direkt eingegeben werden.

Interface
Legt eine neue Schnittstelle an. Der Name kann direkt eingegeben werden. Die erfüllenden und anbietenden Klassen können mit **Provided Interface** und **Requiered Interface** zugeordnet werden.

Constraint
Legt einen Constraint an. Der Constraintname muss über die Properties eingegeben werden.

Attribute
Fügt einer Klasse ein Attribut hinzu. Name und Datentypen werden über die Properties konfiguriert.

Operation
Fügt einer Klasse eine Operation hinzu. Name und Datentypen werden über die Properties konfiguriert.

Enum Literal
Fügt einer **Enumeration** ein neues EnumerationLiteral hinzu.

Association
Legt eine neue Assoziation an. Durch Auswahl des kleinen Pfeilsymboles erscheint folgende Unterauswahl, mit der weitere Elemente angelegt werden können:

Shared Aggregation: Legt eine neue Aggregation an

Composite Aggregation: Legt eine neue Komposition an

Navigable Association: Legt eine navigierbare Assoziation an

- **Dependency**

Legt eine neue Abhängigkeitsbeziehung an. Durch Auswahl des kleinen Pfeilsymboles erscheint folgende Unterauswahl, mit der weitere Elemente angelegt werden können:

Abstraction: Legt eine neue Abstraktionsbeziehung an

Usage: Legt eine neue Use-Beziehung an

Substitution: Legt eine neue Substitute-Beziehung an

Generalization
Legt eine neue Vererbungsbeziehung an. Die Vererbung wird durch Ziehen einer Verbindung von der erbenden Klasse zur Oberklasse angelegt.

Provided Interface
Bietet ein Interface an. Angelegt wird dieses durch Ziehen einer Verbindung von einer Klasse zu einem Interface. Dieses muss vorher mit dem **Interface**-Werkzeug angelegt werden.

Required Interface
Ein Interface nutzen. Angelegt wird dieses durch Ziehen einer Verbindung von einer Klasse zu einem Interface. Dieses muss vorher mit dem **Interface**-Werkzeug angelegt werden.

Constrained Element
Einem Element einen Constraint zuordnen. Der Constraint muss vorher mit dem **Constraint**-Werkzeug angelegt werden.

Association End
Verbindet eine **Association Class** mit einer Assoziation oder einer Klasse.

Realization
Legt eine neue Realisierung an. Die Realisierung wird in Richtung der realisierenden Klasse gezogen. Der Name der Realisierung kann direkt eingegeben werden.

> **Warnung:** Die Elemente **Package**, **DataType**, **Stereotype** und **NAry Dependency** können nicht über die Palette angelegt werden. Die entsprechenden Werkzeuge sind ausgegraut. Für das Anlegen von **Package**, **DataType** und **Stereotype** ist das Kontextmenü der Outline zuständig. **NAry Dependency** werden noch nicht unterstützt.

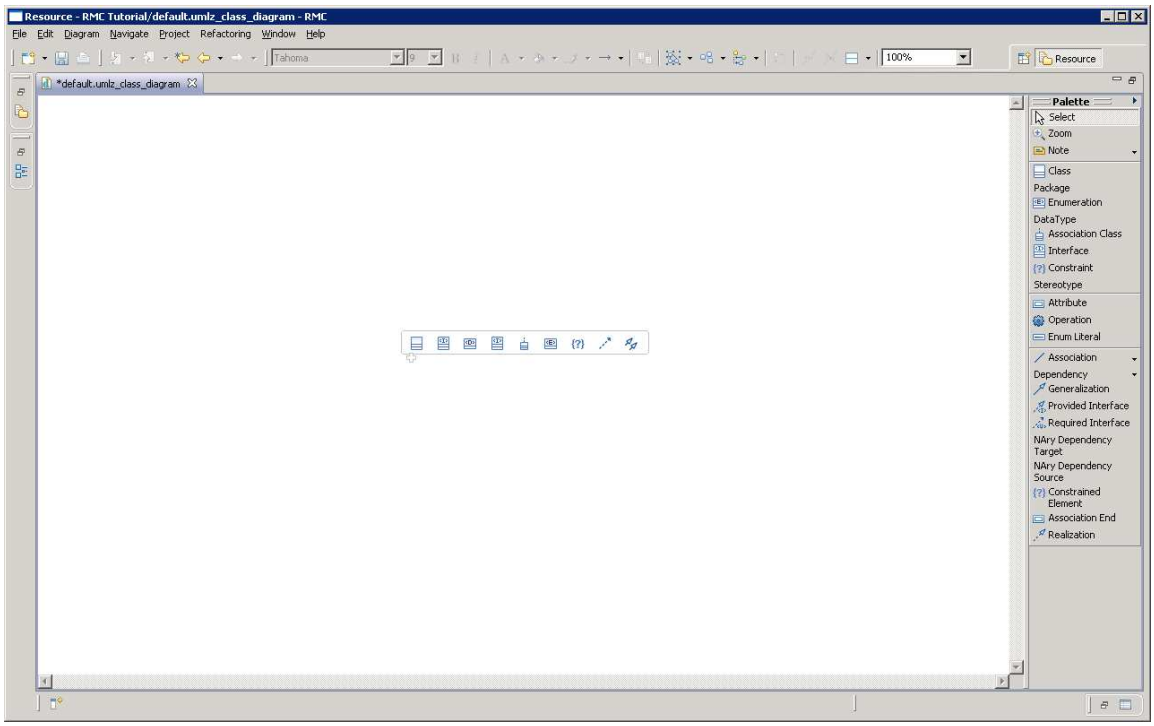


Abbildung 4.11: Popupmenü über dem Canvas des Klassendiagrammeditors

Anlegen von Elementen über den Canvas

Eine weitere Möglichkeit Elemente anzulegen, besteht über den Canvas . Bewegt der Anwender den Mauszeiger auf den Canvas oder über ein Element auf diesem, erscheint nach einigen Sekunden ohne Bewegung ein Menü. Abbildung 4.11 zeigt dieses Popup.

Die Auswahl an Elementen, die erstellt werden können ist Kontextsensitiv. Der Anwender bekommt nur mögliche Auswahlen präsentiert. Die dargestellten Icons sind analog zu der Beschreibung der Palette auf Seite 42.

Anlegen von Elementen über die Outline

Die Outline stellt die dritte Möglichkeit dar, Elemente zu erstellen. Der Wurzelknoten des Baumes ist mit dem selektierten Element auf dem Canvas identisch. Wenn kein Element ausgewählt wurde, ist das **Package** die Wurzel des Baumes. Das Kontextmenü der Outline bietet folgende Unterpunkte:

RMC: Auswahl wichtiger Elemente, die als Kinder des selektierten Elementes angelegt werden können. Dieser Menüpunkt bietet die einfachste Möglichkeit Elemente anzulegen.

Create child: Bietet die Möglichkeit alle Kindelemente des selektierten Elementes anzulegen.

Create sibling: Bietet die Möglichkeit alle Geschwisterelemente des selektierten Elementes anzulegen.

delete: Löscht das ausgewählte Element.

Initialize State Diagram: Legt ein neues Zustandsdiagramm in einer Klasse an.

> **Hinweis:** Der Anwender sollte bevorzugt das ⟨RMC⟩-Menü nutzen. Die beiden Menüpunkte Create child und Create sibling sind nur der Vollständigkeit halber vorhanden. Durch ihre große Anzahl an Elementen bieten sie jedoch wenig Komfort.

Je nach ausgewähltem Element in der Outline sind nicht alle der oben erläuterten Menüeinträge möglich bzw. sichtbar. Einige Elemente können nur über die Outline angelegt werden:

- Type Declaration
- ZExpression Specification
- ZPredicate Specification
- Stereotype

Bearbeiten von Elementen

Dieser Abschnitt erläutert die grundlegenden Möglichkeiten der Bearbeitung von Elementen. Es existieren verschiedene Möglichkeiten, die Eigenschaften von Elementen zu editieren. Diese Optionen werden hier vorgestellt, auf die inhaltlichen Eigenschaften der Elemente wird nicht weiter eingegangen.

Eigenschaften von Elementen

Wählt der Benutzer in der Outline oder im Canvas ein Element aus, erscheinen in der **Property View** die Attribute des Elementes. Abbildung 4.12 zeigt diese Attribute. In diesen kann der Anwender die Eigenschaften des Elementes betrachten und ändern. Es gibt 3 unterschiedliche Typen von Eingabefeldern:

- eine Texteingabe
- eine Auswahl über ein Dropdownmenü
- einen [...]-Button, der einen externen Editor startet

Z-Ausdrücke und Assoziationen lassen sich wie folgt bearbeiten:

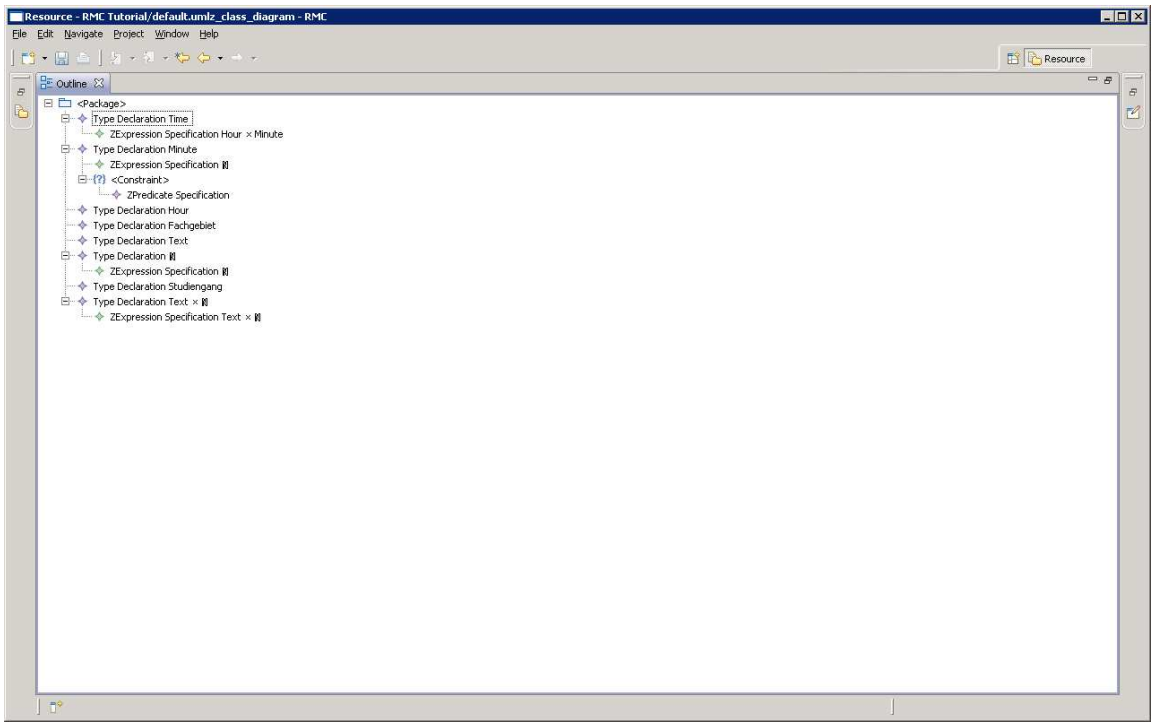


Abbildung 4.12: Properties eines Elementes im Klassendiagrammeditor

Kardinalitäten an Assoziationen: Kardinalitäten werden über die Outline und die Properties bearbeitet. Ungerichtete Assoziationen haben zwei Kindelemente vom Typ **Property**. In diesen werden über **Literal**-Elemente die untere und obere Grenze definiert. Unbeschränktheit wird durch -1 eingegeben[2]. Gerichtete Assoziationen werden ähnlich bearbeitet. Jedoch findet sich hier das Property-Element für den Ursprung in der Klasse und nicht an der Assoziation selber.

Z Ausdrücke: **ZExpression Specifications** und **ZPredicate Specifications** werden über den Z-Editor eingegeben. Dieser ist in Kapitel 4.2.2.2 beschrieben.

Grafisches Editieren

Die Endpunkte aller „Verbindungen“ im Diagramm lassen sich mit der Maus verschieben. Dazu selektiert der Anwender das Ende der Verbindung und schiebt es auf das neue Element. Ebenso ist es möglich, den Typ einer Assoziation nachträglich zu ändern. Dies geschieht über das Kontextmenü der Assoziation ⟨`Association type`⟩.

> **Hinweis:** Bei vielen Elementen ist es oft auch möglich, Texte direkt zu editieren, indem der Benutzer auf das entsprechende Textfeld im Canvas klickt.

[2] Im Diagramm wird diese mit dem üblichen Symbol $*$ dargestellt

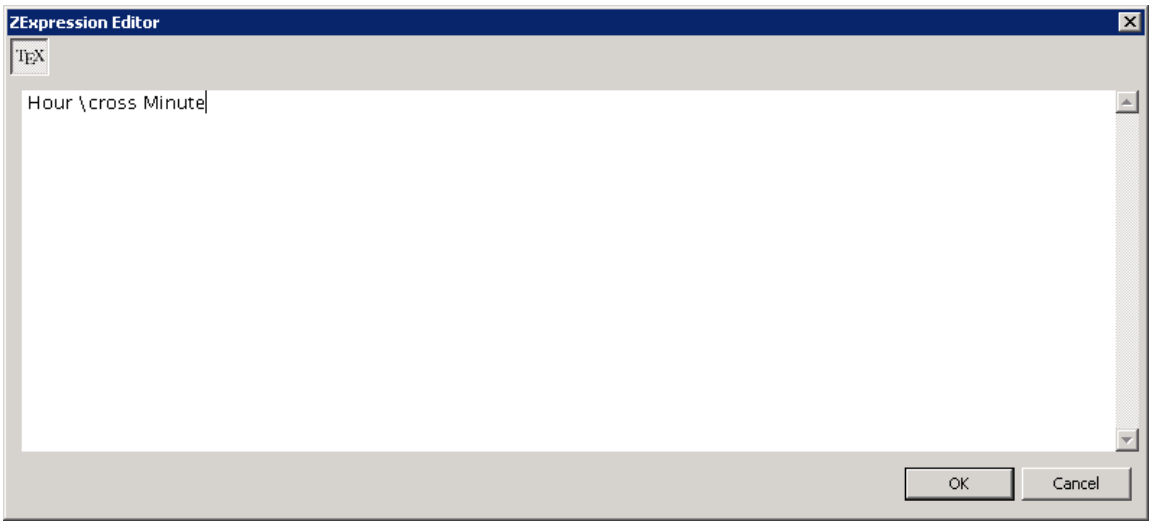


Abbildung 4.13: Eingabe von Z-Ausdrücken im Klassendiagrammeditor

Löschen von Elementen

Der Anwender kann Elemente über den Canvas löschen. Dazu werden ein oder mehrere Elemente selektiert, das Kontextmenü geöffnet und ⟨`Delete from Model`⟩ gewählt. Alternativ können Elemente in der Outline gelöscht werden, indem dort ⟨`delete`⟩ aus dem Kontextmenü gewählt wird.

Die Eingabe von Z Ausdrücken

In UML/Z sind folgende Elemente definiert, die über den Z-Editor bearbeitet werden: **ZPredicate Specification** und **ZExpression Specification**. Diese werden über die Ouline angelegt. Um sie zu bearbeiten, selektiert der Benutzer das gewünschte Element in der Outline. In der **Property View** wird die Bearbeitung durch einen Klick auf den [...]-Button gestartet. Abbildung 4.13 zeigt den geöffneten Editor für die Eingabe und die Bearbeitung von Z-Ausdrücken.

Die Eingabe der Z-Ausdrücke geschieht in LaTeX-Syntax. Die Eingabe wird mit [OK] beendet. Folgende Befehle werden von RMC unterstützt:

Tabelle 4.1: Z-Referenz: Prelude

LaTeX	Unicode	Ausgabe	Bedeutung
`\{`	007B	{	Open bracket
`\}`	007D	}	Close bracket
`\where`	007C		Box separator
`\Delta`	0394	Δ	Schema name prefix
`\Xi`	039E	Ξ	Schema name prefix

LaTeX	Unicode	Ausgabe	Bedeutung
\theta	03B8	θ	Binding expression
\mu	03BC	μ	Definite description
\ldata	27EA	$\langle\!\langle$	Freetype left bracket
\rdata	27EB	$\rangle\!\rangle$	Freetype right bracket
\lblot	2989	$\{\!\!\{$	Binding left bracket
\rblot	298A	$\}\!\!\}$	Binding right bracket
\vdash	22A2	$\vdash$	Conjecture
\land	2227	$\land$	Conjunction
\lor	2228	$\lor$	Disjunction
\implies	21D2	$\Rightarrow$	Implication
\iff	21D4	$\Leftrightarrow$	Equivalence
\lnot	00AC	$\neg$	Negation
\forall	2200	$\forall$	Universal quantifier
\exists	2203	$\exists$	Existential quantifier
\in	2208	$\in$	Set membership
\spot	2981	$\bullet$	Expression separator
@	2981	$\bullet$	Expression separator
\hide	29F9	$\setminus$	Schema hiding
\project	2A21	$\upharpoonright$	Schema projection
\semi	2A1F	$\fatsemi$	Schema composition
\pipe	2A20	$\gg$	Schema piping
\IF	“IF”	if	Conditional
\THEN	“THEN”	then	
\ELSE	“ELSE”	else	
\LET	“LET”	let	Let expression
\pre	“pre”	pre	Schema precondition
\function	“function”	function	Functional operators
\generic	“generic”	generic	Generic operators
\relation	“relation”	relation	Relational operators
\leftassoc	“leftassoc”	*leftassoc*	Left-associative
\rightassoc	“rightassoc”	*rightassoc*	Right-associative
\listarg	“,,”	, ,	List of arguments
\varg	“_”	_	Operator argument
\power	2119	$\mathbb{P}$	Power set
\cross	00D7	$\times$	Cross product
\arithmos	-0001D538	$\mathbb{A}$	Any number
\nat	2115	$\mathbb{N}$	Natural numbers
\alpha	03B1	α	alpha
\beta	03B2	β	beta
\gamma	03B3	γ	gamma
\delta	03B4	δ	delta

LaTeX	Unicode	Ausgabe	Bedeutung
\epsilon	03B5	ϵ	epsilon
\zeta	03B6	ζ	zeta
\eta	03B7	η	eta
\iota	03B9	ι	iota
\kappa	03BA	κ	kappa
\nu	03BD	ν	nu
\xi	03BE	ξ	xi
\pi	03C0	π	pi
\rho	03C1	ρ	rho
\sigma	03C3	σ	sigma
\tau	03C4	τ	tau
upsilon	03C5	υ	upsilon
\phi	03C6	ϕ	phi
\chi	03C7	χ	chi
\psi	03C8	ψ	psi
\omega	03C9	ω	omega
\Gamma	0393	Γ	Gamma
\Theta	0398	Θ	Theta
\Lambda	039B	Λ	Lambda
\Pi	03A0	Π	Pi
\Sigma	03A3	Σ	Sigma
\Upsilon	03A5	Υ	Upsilon
\Phi	03A6	Φ	Phi
\Psi	03A8	Ψ	Psi
\Omega	03A9	Ω	Omega

Tabelle 4.2: Z-Referenz: Number Toolkit

LaTeX	Unicode	Ausgabe	Bedeutung
succ	“succ”	*succ*	Successor function
\num	2124	$\mathbb{Z}$	Integers
\negate	002D	-	Arithmetic negation
-	“-”	$-$	Subtraction
\leq	2264	$\leq$	Less than or equal
<	“<”	$<$	Less than
\geq	2265	$\geq$	Greater than or equal
>	“<”	$>$	Greater than
\nat_1		$\mathbb{N}_1$	Strictly positive $\mathbb{N}$
*	“*”	$*$	Multiplication
\div	“div”	div	Division
\mod	“mod”	mod	Modulus

Tabelle 4.3: Z-Referenz: Set Toolkit

LaTeX	Unicode	Ausgabe	Bedeutung
\rel	2194	$\leftrightarrow$	Relations
\fun	2192	$\rightarrow$	Total functions
\neq	2260	$\neq$	Inequality
\notin	2209	$\notin$	Non-membership
\emptyset	2205	$\varnothing$	Empty set
\subseteq	2286	$\subseteq$	Subset relation
\subset	2282	$\subset$	Proper subset
\power_1		$\mathbb{P}_1$	Non-empty subsets
\cup	222A	$\cup$	Set union
\cap	2229	$\cap$	Set intersection
\setminus	005C	$\setminus$	Set difference
\symdiff	2296	$\ominus$	Set symmetric difference
\bigcup	22C3	$\bigcup$	Generalised union
\bigcap	22C2	$\bigcap$	Generalised intersection
\finset	-0001D53D	$\mathbb{F}$	Finite subsets
\finset_1		$\mathbb{F}_1$	Non-empty finite subsets

Tabelle 4.4: Z-Referenz: Relation Toolkit

LaTeX	Unicode	Ausgabe	Bedeutung
first	“first”	*first*	Tuple projection
second	“second”	*second*	Tuple projection
\mapsto	21A6	$\mapsto$	Maplets
\dom	“dom”	dom	Domain
\ran	“ran”	ran	Range
\id	“id”	id	Identity relation
\comp	2A3E	$\fatsemi$	Relational composition
\circ	2218	$\circ$	Functional composition
\dres	25C1	$\lhd$	Domain restriction
\rres	25B7	$\rhd$	Range restriction
\ndres	2A64	$\mathbin{⩤}$	Domain subtraction
\nrres	2A65	$\mathbin{⩥}$	Range subtraction
\inv	223C	$\sim$	Relational inversion
\limg	2987	$\limg$	Rel. image left bracket
\rimg	2988	$\rimg$	Rel. image right bracket
\oplus	2295	$\oplus$	Overriding
\plus	“^+”	$^+$	Transitive closure
\star	“^*”	*	Reflexive transitive closure

Tabelle 4.5: Z-Referenz: Function Toolkit

LaTeX	Unicode	Ausgabe	Bedeutung
\pfun	21F8	⇸	Partial functions
\pinj	2914	⤔	Partial injections
\inj	21A3	↣	Total injections
\psurj	2900	⤖	Partial surjections
\surj	21A0	↠	Total surjections
\bij	2916	⤖	Bijections
\ffun	21F8	⇸	Finite functions
\finj	2915	⤔	Finite injections
\disjoint	“disjoint”	disjoint	Disjointness
\partition	“partition”	partition	Partitions

Tabelle 4.6: Z-Referenz: Sequence Toolkit

LaTeX	Unicode	Ausgabe	Bedeutung
\upto	“..”	..	Number range
iter	“iter”	*iter*	Iteration
\#	0023	#	Set cardinality
min	“min”	*min*	Minimum
max	“max”	*max*	Maximum
\seq	“seq”	seq	Finite sequences
\seq_1		seq_1	Non-empty finite sequences
\iseq	“iseq”	iseq	Injective sequences
\langle	27E8	⟨	Sequence left bracket
\rangle	27E9	⟩	Sequence right bracket
\cat	2040	⁀	Sequence concatenation
rev	“rev”	*rev*	Reverse
head	“head”	*head*	Head of sequence
last	“last”	*last*	Last of sequence
tail	“tail”	*tail*	Tail of sequence
front	“front”	*front*	Front of sequence
squash	“squash”	*squash*	Squashing
\extract	21BF	↾	Extracting
\filter	21BE	↾	Filtering
\prefix	“prefix”	*prefix*	Prefix relation
\suffix	“suffix”	*suffix*	Suffix relation
\infix	“infix”	*infix*	Infix relation
\dcat		⁀/	Distributed concatenation

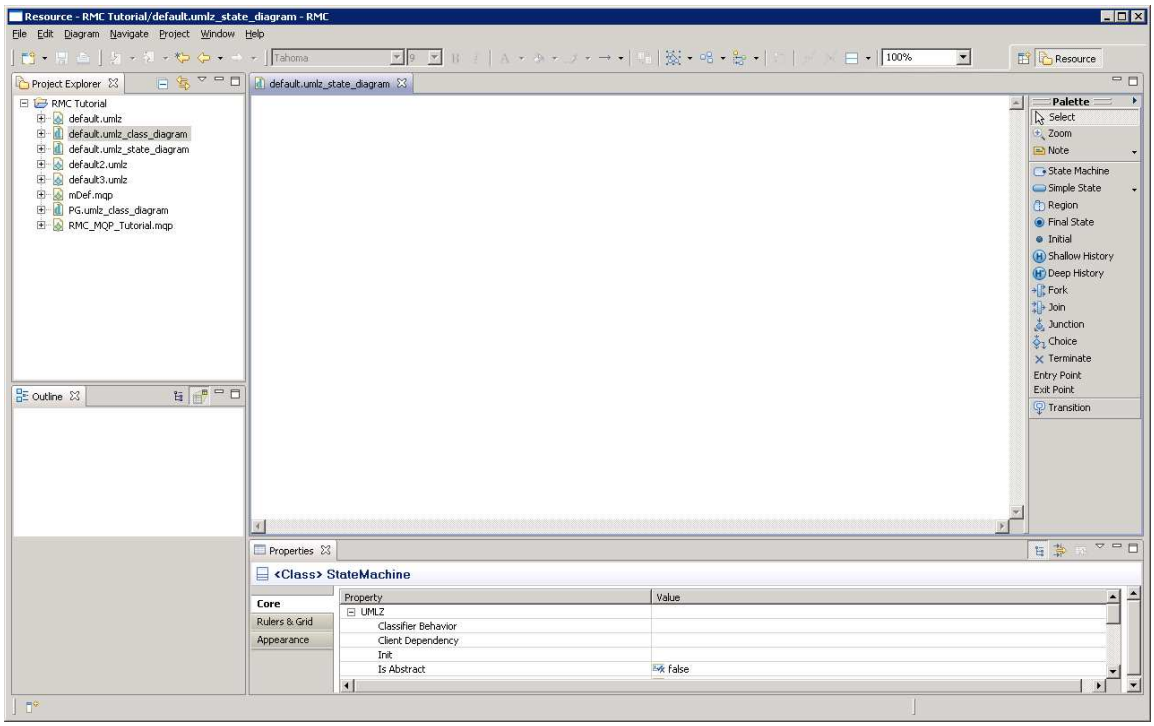


Abbildung 4.14: Übersicht über Zustandsdiagrammeditor

4.2.2.3 Zustandsdiagramme

Zweiter von uns unterstützter Diagrammtyp ist das Zustandsdiagramm. Abbildung 4.14 zeigt den UML/Z-Editor mit einem geöffneten Zustandsdiagramm.

Wird der Editor mit einem Zustandsdiagramm geöffnet, zeigen sich folgende Bereiche:

Canvas: Der Canvas ist die Zeichenfläche, auf dem die Elemente des Diagrammes als Grafiken erscheinen.

Palette: Die Palette am rechten Rand des Canvas ist das Werkzeug, um Elemente anzulegen.

Properties: In den Properties werden Informationen über das Element dargestellt, das gerade selektiert ist. Wenn das Fenster mit den Properties nicht sichtbar ist, kann es mit ⟨`Window`⟩→⟨`Show View`⟩→⟨`Properties`⟩ geöffnet werden.

Outline: Die Outline bietet eine verkleinerte grafische Übersicht über das aktuelle Diagramm. Wenn die Oulineview nicht sichtbar ist, kann sie mit ⟨`Window`⟩→⟨`Show View`⟩→⟨`Outline`⟩ geöffnet werden.

Project Explorer: Der Project Explorer bietet einen Überblick über die Projekte. Die einzelnen Modelle und Diagramme sind aufgelistet.

Hinweis: Falls ein Fenster (in Eclipse-Terminologie „eine **View**“) fehlen sollte: Mit ⟨`Window`⟩→⟨`Show View`⟩ können Fenster neu geöffnet werden.

Anlegen von Elementen über die Palette

Die Elemente eines Zustandsdiagrammes werden mit Hilfe der Palette angelegt. Dazu selektiert der Anwender das gewünschte Element in der Palette und klickt auf den Canvas. Elemente die zwei Knoten verbinden, werden durch Ziehen einer Verbindung zwischen Elementen auf dem Canvas erstellt. Folgende Symbole finden sich in der Palette:

Statemachine
Legt eine neue Statemachine an. Eine Statemachine braucht immer eine Region als Inhalt. Der Name der Statemachine kann direkt nach dem Anlegen eingegeben werden.

Simple State
Legt einen neuen Zustand an. Durch Auswahl des kleinen Pfeilsymboles erscheint folgende Unterauswahl, mit der weitere Elemente angelegt werden können:

Composite State: Legt einen komplexen Zustand an.

Submachine State: Fügt der Region eine weitere Statemachine als Submachine hinzu. Die eingebettete Statemachine wird über einen Dialog erfragt.

Region
Legt eine neue Region an. Eine Region nimmt Zustände und Transitionen auf.

Final State
Legt den Endzustand an. Dieser liegt immer in einer Region.

Initial
Legt den Startzustand an. Dieser liegt immer in einer Region.

Shallow History
Legt eine „flache Historie“ an, um den letzen aktiven Zustand des umschließenden States zu speichern.

Deep History
Legt eine „Historie“ an, um den letzen aktiven Zustand des umschließenden States, aber nicht der Unterzustände, zu speichern.

Fork
Legt eine Gabelung von Transitionen an.

Join
Legt eine Zusammenführung mehrer Transitionen an.

Junction
Legt eine neue Verzweigung von Transitionen an.

Choice
Legt eine neue Entscheidung von Transitionen an.

Terminate
Legt einen Endzustand von Transitionen an.

- **Entry Point**

Legt den Eintrittspunkt der Statemachine an.

- **Exit Point**

Legt den Austrittspunkt der Statemachine an.

Transition
Legt eine neue Transition an. Die Zuordnung von Methoden zu Transitionen ist auf Seite 57 erläutert.

> **Hinweis:** Um einer Statemachine Zustände hinzuzufügen, wird eine **Region** benötigt.

> **Warnung:** Der Canvas stellt genau eine Statemachine dar. Bei Änderung der Größe dieser Statemachine kann es zu unvorhersehbaren Effekten kommen. In diesen Fall hilft es das Diagramm zu schließen und wieder zu öffnen. Teilweise ist es nötig, dass Diagramm neu aus dem Modell zu erstellen.

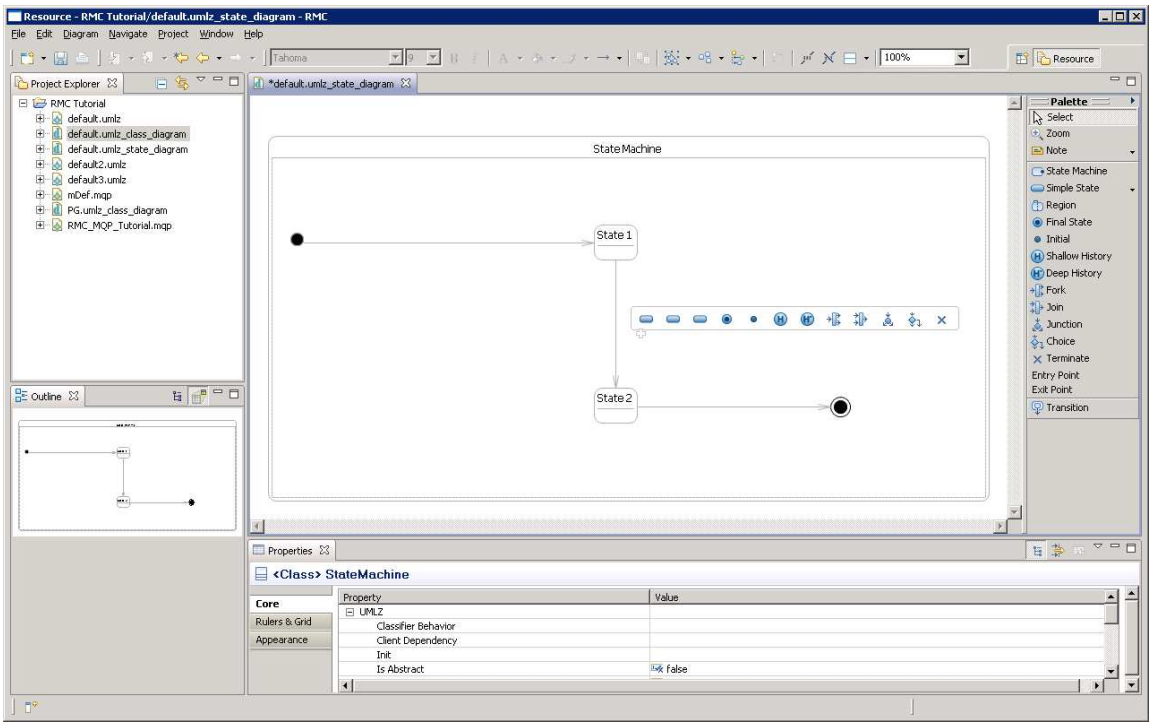


Abbildung 4.15: Popupmenü über dem Canvas des Zustandsdiagrammeditors

Anlegen von Elementen über den Canvas

Eine weitere Möglichkeit Elemente anzulegen besteht über den Canvas. Bewegt der Anwender den Mauszeiger auf den Canvas oder über ein Element auf diesem erscheint nach einigen Sekunden ohne Bewegung ein Menü. Abbildung 4.15 zeigt dieses Popupmenü.

Die Auswahl an Elementen, die erstellt werden können, ist kontextsensitiv. Der Anwender bekommt nur die jeweils möglichen Auswahlen präsentiert. Die dargestellten Icons sind analog zu der Beschreibung der Palette auf Seite 54.

Bearbeiten von Elementen

Dieser Abschnitt erläutert die grundlegenden Möglichkeiten der Bearbeitung von Elementen. Es existieren verschiedene Möglichkeiten die Eigenschaften von Elementen zu editieren. Diese Optionen werden hier vorgestellt, auf die inhaltlichen Eigenschaften der Elemente wird nicht weiter eingegangen.

Eigenschaften von Elementen

Wählt der Benutzer im Canvas ein Element aus, erscheinen in der **Property View** die Attribute des ausgewählten Elementes. Abbildung 4.12 auf Seite 47 in der Beschreibung

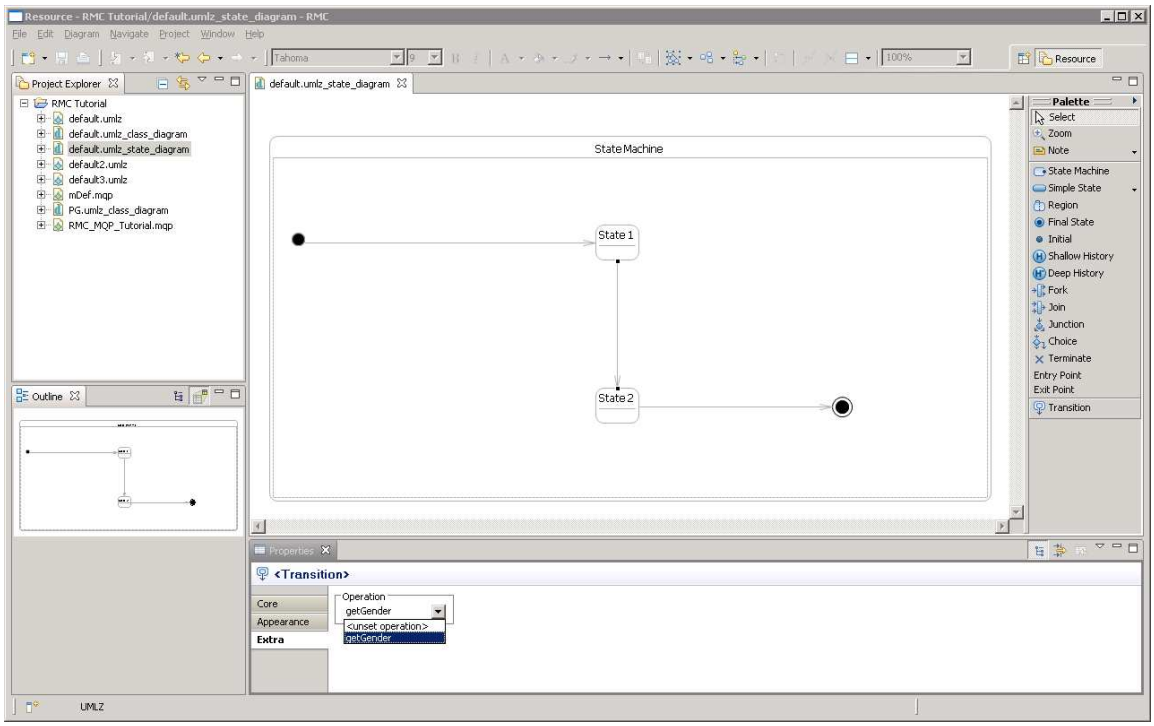


Abbildung 4.16: Zuordnung von Operationen auf Transitionen im Zustandsdiagrammeditor

des Klasseneditors zeigt dieses Fenster. In diesem kann der Anwender die Eigenschaften des Elementes betrachten und bearbeiten. Es gibt drei unterschiedliche Typen von Eingabefeldern:

- eine Texteingabe
- eine Auswahl über ein Dropdownmenü
- einen [...]-Button, der einen externen Editor startet

Transitionen Operationen zuweisen

Um einer Transition eine Operation zuzuorden, wählt der Anwender die entsprechende Transition auf dem Canvas aus. In den **Properties** wird der **Extra-Tab** gewählt. Dieses ist in Abbildung 4.16 dargestellt.

Zugeordnet werden können die entsprechenden Methoden der Klasse, in der die Statemachine enthalten ist, und die Methoden der Superklassen dieser Klasse. Transitionen von **Initial States** können keine Operationen zugeordnet werden.

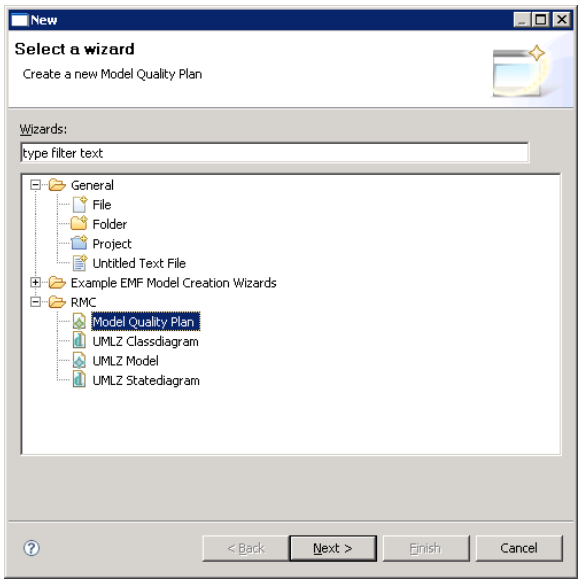


Abbildung 4.17: MQP-Wizard: MQP auswählen

Löschen von Elementen

Der Anwender kann Elemente über den Canvas löschen. Dazu werden ein oder mehrere Elemente selektiert, mittels rechter Maustaste das Kontextmenü geöffnet und ⟨`Delete from Model`⟩ gewählt.

4.2.3 Modellqualitätsplan

Der Editor des Modellqualitätsplans (MQP) ermöglicht die Eingabe eines MQPs. Um ein MQP anlegen zu können, muss vorher ein Projekt in Eclipse vorhanden sein (siehe Kapitel 4.2.1).

4.2.3.1 MQP anlegen

Abbildung 4.18 zeigt den Dialog für den Dateinamen des MQPs und das Projekt, in welchem der neue MQP gespeichert werden soll. Hat man dieses ausgewählt, geht es mit [Next] zum nächsten Schritt.

In Abbildung 4.19 werden die allgemeinen Informationen über den MQP eingegeben. Als erstes muss dafür ein Name für den MQP (**model quality plan name**) und eine Version (**version**) eingetragen werden. Der Name ist dabei frei wählbar. Die Versionsnummer muss in der Form x.y sein, wobei x und y für Zahlen stehen. Des Weiteren werden das Softwareentwicklungsmodell (**software development model**), die Entwicklungsphase (**development phase**) sowie der Status (**status**) abgefragt. In allen drei Fällen hat der Anwender eine Auswahl zur Verfügung.

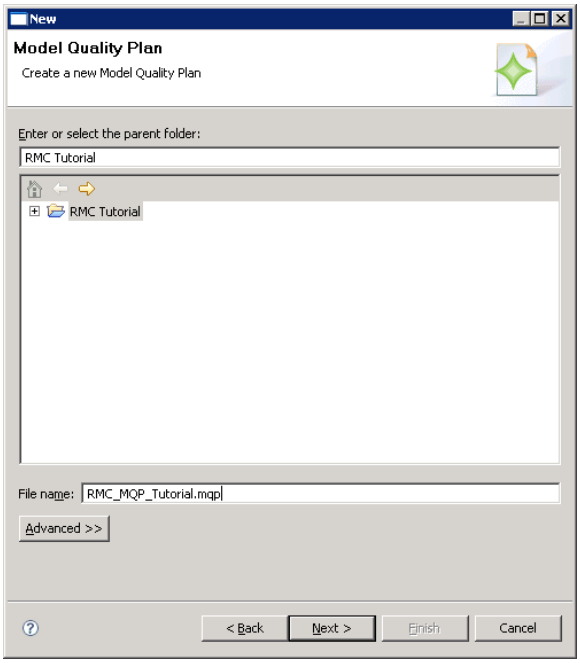


Abbildung 4.18: MQP-Wizard: Name und Pfad des MQPs festlegen

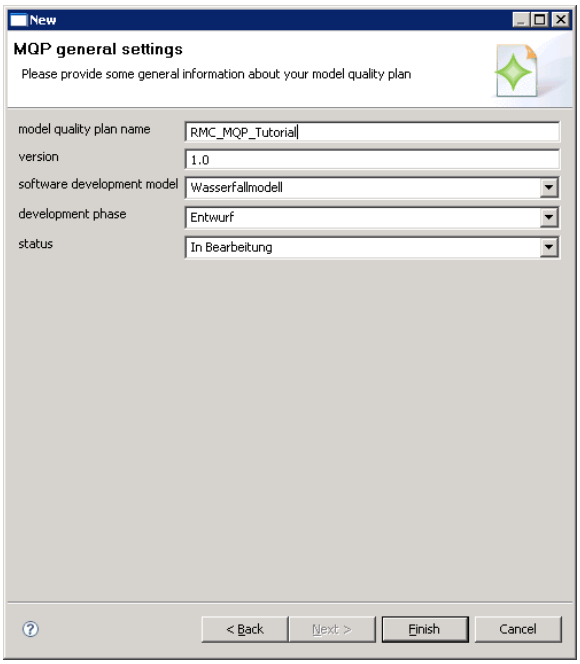


Abbildung 4.19: MQP-Wizard: Generelle MQP-Informationen

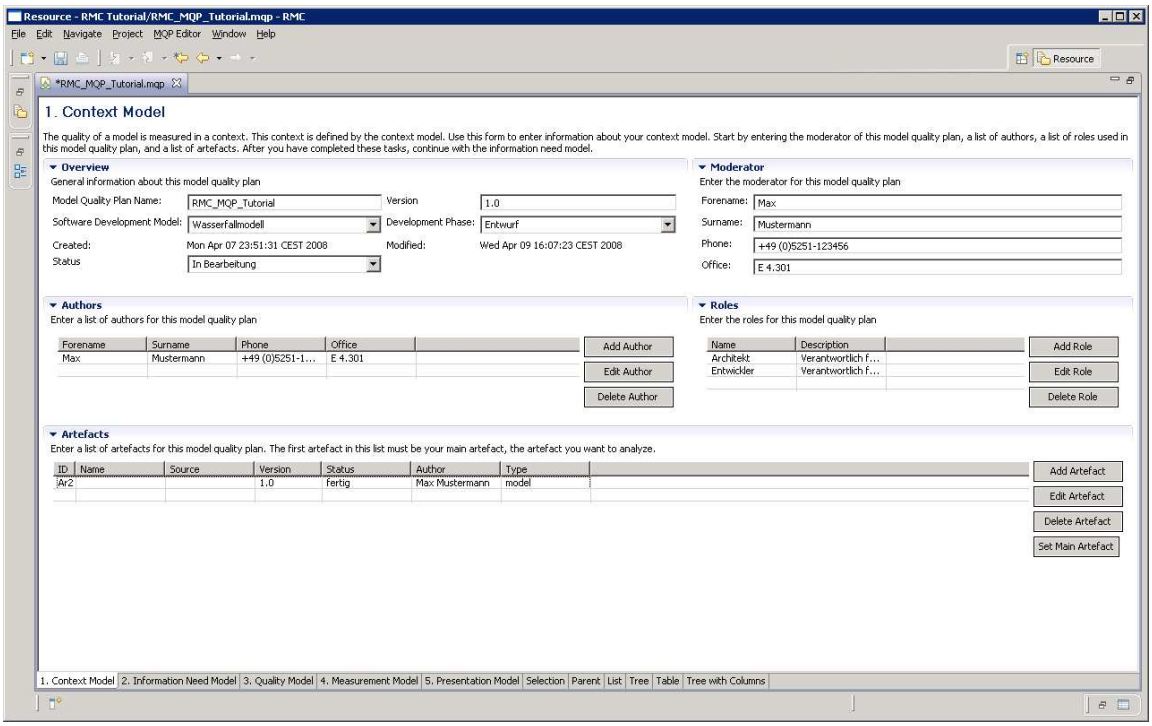

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
*RMC_MQP_Tutorial.mqp
1. Context Model
The quality of a model is measured in a context. This context is defined by the context model. Use this form to enter information about your context model. Start by entering the moderator of this model quality plan, a list of authors, a list of roles used in this model quality plan, and a list of artefacts. After you have completed these tasks, continue with the information need model.
Overview
General information about this model quality plan
Model Quality Plan Name: RMC_MQP_Tutorial
Version: 1.0
Software Development Model: Wasserfallmodell
Development Phase: Entwurf
Created: Mon Apr 07 23:51:31 CEST 2008
Modified: Wed Apr 09 16:07:23 CEST 2008
Status: In Bearbeitung
Moderator
Enter the moderator for this model quality plan
Forename: Max
Surname: Mustermann
Phone: +49 (0)5251-123456
Office: E 4.301
Authors
Enter a list of authors for this model quality plan
Forename Surname Phone Office
Max Mustermann +49 (0)5251-1... E 4.301
Add Author
Edit Author
Delete Author
Roles
Enter the roles for this model quality plan
Name Description
Architekt Verantwortlich f...
Entwickler Verantwortlich f...
Add Role
Edit Role
Delete Role
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
Ar2 1.0 fertig Max Mustermann model
Add Artefact
Edit Artefact
Delete Artefact
Set Main Artefact
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns

Abbildung 4.20: MQP-Editor: Kontextmodell

Mit [Finish] wird ein MQP angelegt, und eine Konfigurationsdatei mit bereits vordefinierten Messungen in das Projekt kopiert. Der MQP öffnet sich automatisch und der erste Reiter, das Kontextmodell (siehe Abbildung 4.20) wird angezeigt.

4.2.3.2 Der MQP-Editor

Um einen MQP anzulegen, gibt es einen eigenen Editor. Ein MQP besteht dabei aus fünf Modellen:

- Kontextmodell (**Context Model**)
- Informationsbedürfnismodell (**Information Need Model**)
- Qualitätsmodell (**Quality Model**)
- Messmodell (**Measurement Model**)
- Präsentationsmodell (**Presentation Model**)

Für jedes Model gibt es im MQP einen Reiter, die sich unterhalb des Modells befinden. Die Reihenfolge der Tabs entspricht auch der Reihenfolge, in der der Benutzer die Informationen eingeben sollte, da sich die Modelle aufeinander beziehen. Es ist dem Anwender jedoch möglich jederzeit zwischen den einzelnen Modellen hin und her zu springen, und Eingaben zu korrigieren oder zusätzlich zu tätigen.

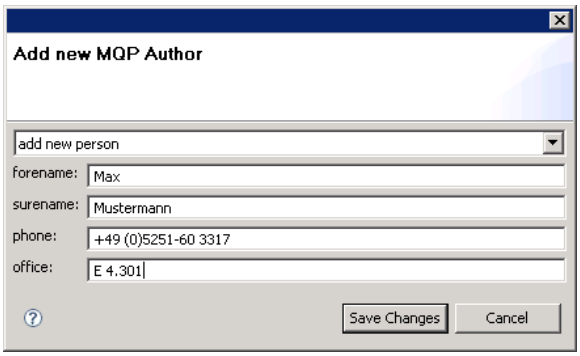


Abbildung 4.21: MQP-Kontextmodell: Neuen MQP-Autor hinzufügen

4.2.3.3 MQP-Kontextmodell

Das Kontextmodell besteht wiederum aus fünf Bereichen, die im Folgenden genauer beschrieben werden.

Overview-Bereich

Im Overview-Bereich werden die allgemeinen Informationen über den MQP, die beim Anlegen bereits eingegeben wurden, angezeigt und können dort auch verändert werden.

Moderator-Bereich

Ein MQP besitzt einen Moderator, der die Verantwortung für den MQP trägt. Einem Moderator kann der Vorname (**forename**), der Nachname (**surname**), die Telefonnummer (**phone**) sowie das Büro (**office**) zugewiesen werden. Die Eingaben werden automatisch gespeichert, sobald etwas in die Zeilen eingegeben wurde.

Autoren-Bereich

An der Erstellung eines MQPs können mehrere Autoren beteiligt sein. Über [Add Author] (Abbildung 4.21) kann der Anwender neue Autoren anlegen. Im sich neu öffnenden Fenster kann der Benutzer auswählen, ob eine neue Person angelegt werden, oder ob der bereits eingegebene Moderator auch als Autor hinzugefügt werden soll. Soll ein neuer Autor angelegt werden, kann der Benutzer den Vornamen (**forename**), den Nachnamen (**surname**), die Telefonnummer (**phone**) sowie das Büro (**office**) eintragen und die Eingaben mit [Save Changes] speichern.

Die zu einem MQP gehörenden Autoren werden im Autoren-Bereich in einer Tabelle dargestellt. Um einen Autor zu bearbeiten, muss der Benutzer den entsprechenden Autor in der Tabelle auswählen und [Edit Author] drücken. Im neuen Fenster kann der Benutzer die

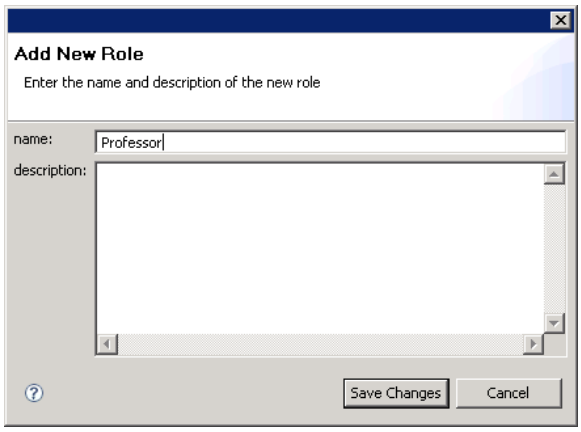


Abbildung 4.22: MQP-Kontextmodell: Neue Rolle

Eingaben editieren und über [Save Changes] speichern. Um einen Autor zu löschen, muss der Anwender den Autor in der Tabelle auswählen und über [Delete Author] löschen.

Rollen-Bereich

Ein MQP kann für verschiedene Personen interessant sein. Dafür gibt es im MQP die Rollen. Im Rollen-Bereich kann der Anwender Rollen anlegen [Add Role], Rollen editieren [Edit Role] und Rollen löschen [Delete Role]. Um eine Rolle anzulegen (Abbildung 4.22), muss der Benutzer einen Namen sowie optional eine Beschreibung eingeben. Über [Save Changes] speichert man die Rolle. Die gespeicherten Rollen werden in einer Tabelle angezeigt.

Artefakt-Bereich

Im Artefakt-Bereich können Artefakte zum MQP hinzugefügt werden. Die hinzugefügten Artefakte werden wie bei den Autoren und Rollen in einer Tabelle aufgelistet. Über [Add Artefact], [Edit Artefact] oder [Delete Artefact] kann der Benutzer Artefakte anlegen, editieren oder löschen. Ein MQP hat immer ein Hauptartefakt vom Typ Softwaremodell, auf das sich der MQP bezieht. Dieses erscheint immer als Erstes in der Liste. Daher ist es zuerst auch nur möglich, ein Artefakt vom Typ Softwaremodell anzulegen, bevor der Benutzer andere Arten von Artefakte hinzufügen kann. Sollte sich das Hauptartefakt ändern, kann ein anderes Artefakt (aber auch nur vom Typ Softwaremodell) in der Tabelle ausgewählt und über [Set Main Artefact] zum neuen Hauptartefakt gemacht werden. Dadurch wird das ausgewählte Artefakt automatisch an die erste Stelle in der Tabelle gerückt.

Um ein Artefakt anlegen zu können (siehe Abbildung 4.23), werden verschiedene Informationen benötigt. Im oberen Teil des Dialogs werden die Artefaktinformationen (**arefact information**) eingegeben. Dazu gehört eine ID, eine Phase, eine Versionsnummer und ein Status. Der Name und die Quelle (Source) sind ausgegraut, da sie später automatisch vom Tool nach den Messungen im Messmodell eingetragen werden. Falls sich der Benutzer die

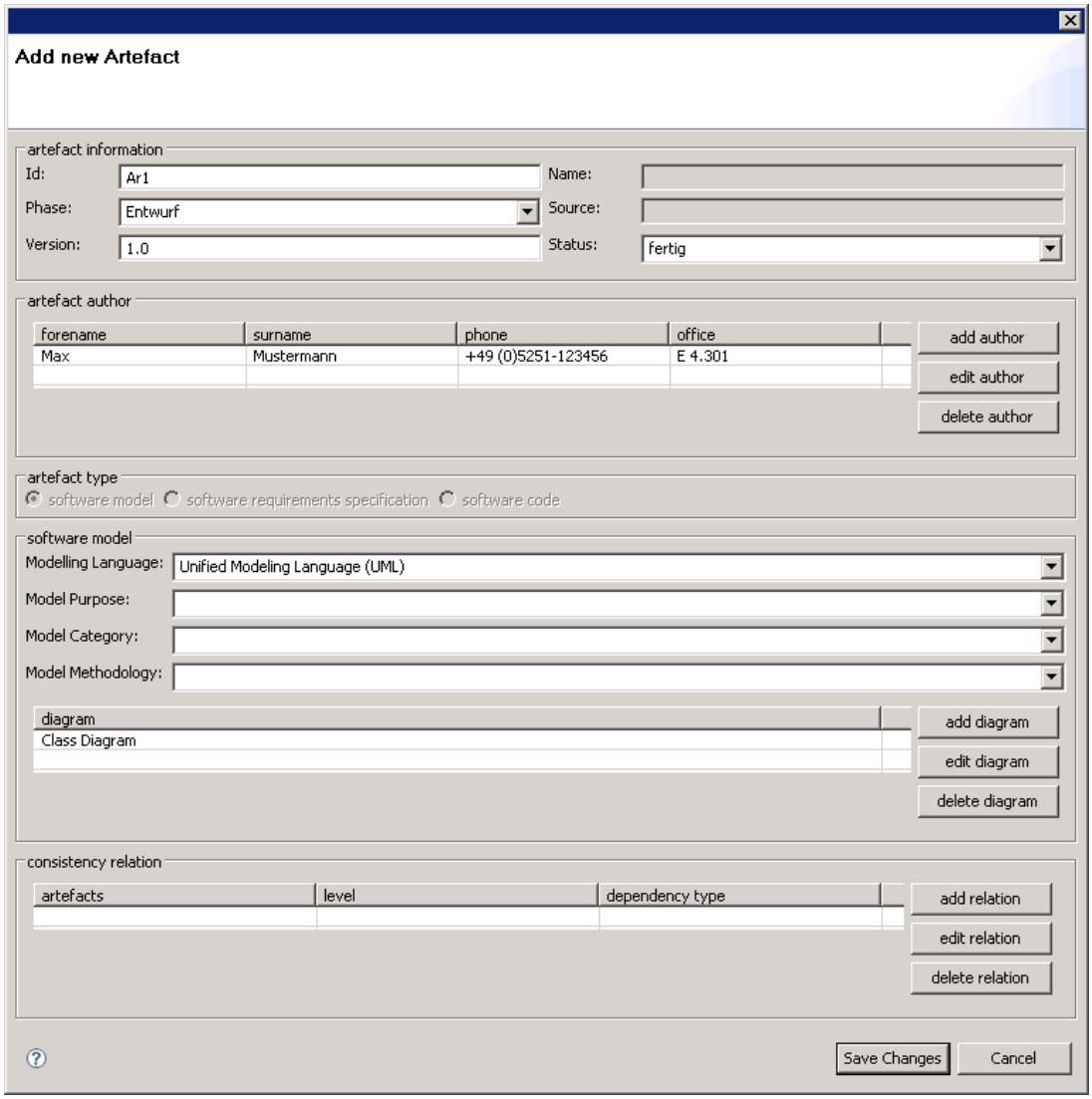


Abbildung 4.23: MQP-Kontextmodell: Neues Artefakt hinzufügen

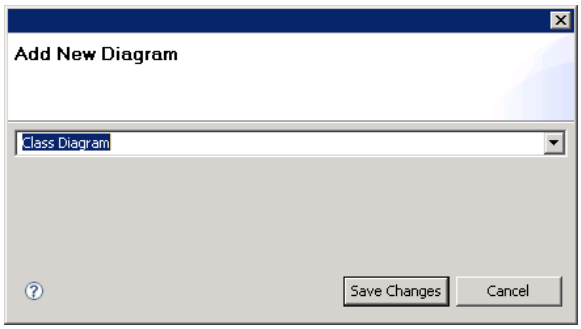


Abbildung 4.24: MQP-Neues Artefact: Neues Diagram hinzufügen

Artefakte nach den Messungen nochmals ansehen will, würde hier der Name und die Quelle stehen.

Zu einem Artefakt können einer oder mehrere Autoren gehören. Diese können unter den Artefaktinformationen im Bereich **artefact author** über [add author], [edit author] und [delete author] angelegt, editiert und gelöscht werden. Dies erfolgt analog zum Anlegen, Editieren und Löschen eines Autors im Kontextmodell. Beim Anlegen eines Autors kann der Benutzer eine neue Person hinzufügen, oder einen bereits im MQP angelegte Person auswählen (siehe Abbildung 4.21). Die hinzugefügten Autoren werden in einer Tabelle angezeigt. Die Reihenfolge hat keine Bedeutung.

Ein Artefakt kann vom Typ Softwaremodell (**software model**), Spezifikation (**software requirements specification**) oder Code (**software code**) sein. Dies kann unter den Autoren im Bereich Artefakttyp (**artefact type**) ausgewählt werden. Wurde noch kein Artefakt im MQP angelegt, ist automatisch der Typ Softwaremodell ausgewählt und kann auch nicht geändert werden. Wurde bereits ein Artefakt vom Typ Softwaremodell angelegt, können auch die anderen Typen ausgewählt werden.

Abhängig vom Typ kann der Benutzer noch optionale Informationen eingeben. Diese zusätzlichen Informationen werden im Bereich unter der Auswahl des Typs abgefragt. Beim Typ Softwaremodell kann der Anwender die Modellsprache (**Modelling Language**), die Modellabsicht (**Model Purpose**), die Modellkategorie (**Model Category**) und die Modellmethodik (**Model Methodology**) auswählen. Je nach Konfiguration des MQPs können bereits vordefinierte Einträge zum Beispiel in Abbildung 4.23 bei Modellsprache **Unified Modeling Language (UML)** ausgewählt, oder Werte eingetragen werden. Die Softwaremodelle können verschiedene Arten von Diagrammen zum Beispiel Klassendiagramm (**Class Diagram**) enthalten, welche optional von Benutzer hinzugefügt ([add diagram], siehe Abbildung 4.24), bearbeitet ([edit diagram]) oder gelöscht ([delete diagram]) werden können. Die hinzugefügten Diagrammarten werden in einer Tabelle angezeigt.

Wird als Artefakttyp Spezifikation (**software requirements specification**) gewählt, kann der Benutzer dort die Spezifikationssprache (**specification language**) und ein Formalisierungslevel (**formalization level**) angeben. Über [add language], [edit language] und [delete language] kann der Anwender Spezifikationen anlegen (siehe Abbildung 4.25), bearbeiten oder löschen. Die angelegten Spezifikationen werden in einer Tabelle angezeigt.

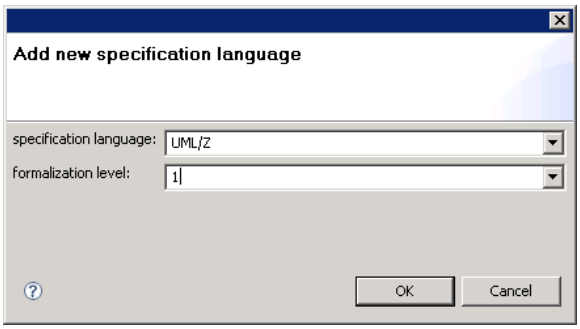


Abbildung 4.25: MQP-Neues Artefact: Neue Spezifikation hinzufügen

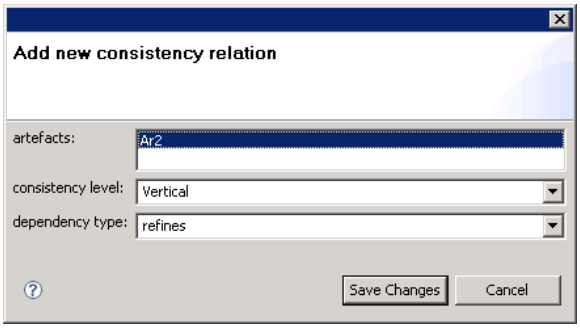


Abbildung 4.26: MQP-Neues Artefact: Neue Beziehung zu einem anderen Artefakt anlegen

Beim letzten Artefakttyp Code (**software code**) kann der Anwender die Programmiersprache (**Programming Language**) angegeben. Dabei stehen dem Benutzer einige Programmiersprachen wie C# oder Java zur Auswahl.

Als letzten Schritt kann der Benutzer noch Beziehungen (**consistency relations**) zu anderen Artefakten anlegen [add relation], bearbeiten [edit relation] oder löschen [delete relation]. Um eine Beziehung anzulegen (siehe Abbildung 4.26), muss der Benutzer ein Artefakt, auf das sich das neu anzulegende Artefakt beziehen soll, einen Beziehungslevel (**consistency level**) und einen Beziehungstyp (**dependency type**) auswählen. Das Beziehungslevel gibt an, ob die Beziehung zum Beispiel innerhalb einer Entwicklungsphase (horizontal) oder zwischen den verschiedenen Phasen (vertikal) ist. Der Beziehungstyp gibt zum Beispiel an, ob das Dokument ein anderes verfeinert oder davon abstrahiert. Über [Save Changes] kann der Anwender die Beziehung speichern.

Abschließend speichert der Anwender das Artefakt über [Save Changes]. Das angelegte Artefakt erscheint jetzt im Kontextmodell in der Artefakttabelle. Ein Beispiel für ein ausgefülltes Kontextmodell zeigt die Abbildung 4.27.

4.2.3.4 MQP-Informationsbedürfnismodell

Um vom Kontextmodell ins **Informationsbedürfnismodell** zu gelangen, muss der Benutzer den Reiter **Information Need Model** auswählen. Das Informationsbedürfnismodell (siehe Abbildung 4.28) besteht aus zwei Bereichen. Einen Fragen- (**Question**) und einen

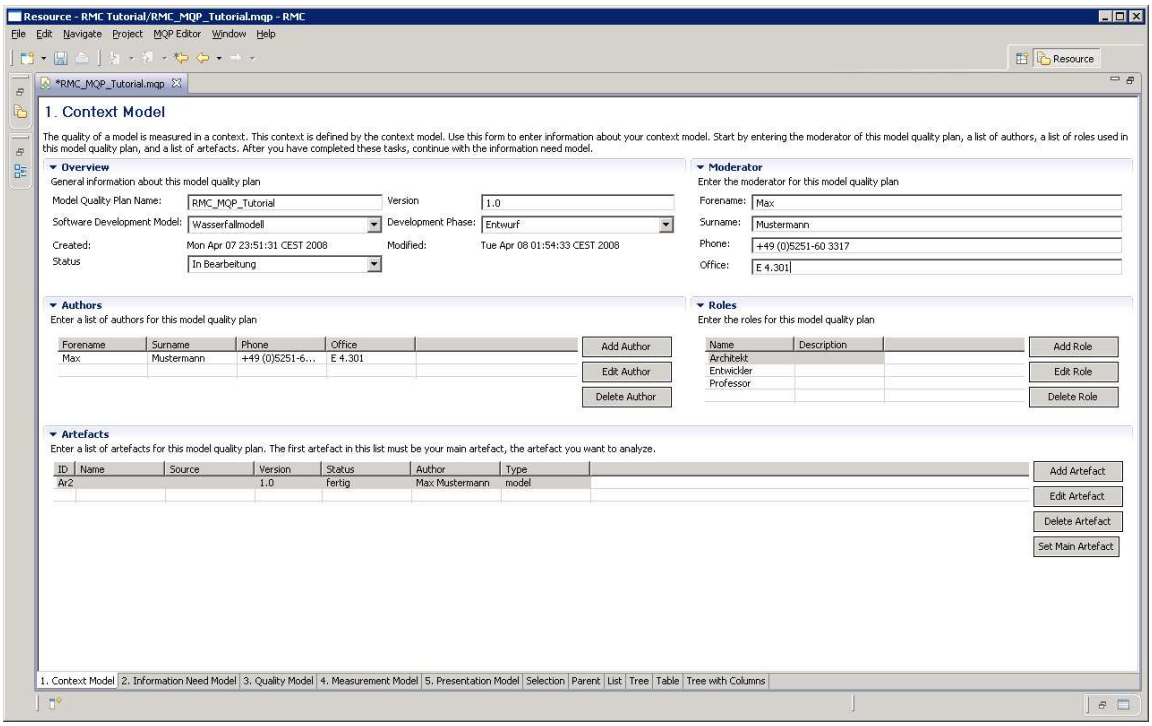

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
*RMC_MQP_Tutorial.mqp
1. Context Model
The quality of a model is measured in a context. This context is defined by the context model. Use this form to enter information about your context model. Start by entering the moderator of this model quality plan, a list of authors, a list of roles used in this model quality plan, and a list of artefacts. After you have completed these tasks, continue with the information need model.
Overview
General information about this model quality plan
Model Quality Plan Name: RMC_MQP_Tutorial
Version: 1.0
Software Development Model: Wasserfallmodell
Development Phase: Entwurf
Created: Mon Apr 07 23:51:31 CEST 2008
Modified: Tue Apr 08 01:54:33 CEST 2008
Status: In Bearbeitung
Moderator
Enter the moderator for this model quality plan
Forename: Max
Surname: Mustermann
Phone: +49 (0)5251-60 3317
Office: E 4.301
Authors
Enter a list of authors for this model quality plan
Forename Surname Phone Office
Max Mustermann +49 (0)5251-6... E 4.301
Add Author
Edit Author
Delete Author
Roles
Enter the roles for this model quality plan
Name Description
Architekt
Entwickler
Professor
Add Role
Edit Role
Delete Role
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
Ar2 1.0 fertig Max Mustermann model
Add Artefact
Edit Artefact
Delete Artefact
Set Main Artefact
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns

Abbildung 4.27: MQP-Editor: ausgefülltes Kontextmodell

Artefaktbereich (**Artefacts**). Im Artefaktbereich werden als Unterstützung für den Benutzer die Artefakte aufgelistet, die bereits im Kontextmodell angelegt wurden.

Im Fragenbereich (Question) kann der Benutzer Fragegruppen (**Questiongroup**), Fragen (**Question**) und Ziele (**Goal**) anlegen oder löschen. Durch Klicken auf [add Goal] erscheint ein Eingabefenster (siehe Abbildung 4.29), in dem der Benutzer das Ziel und ein **Goal Purpose** auswählen kann. Bei einer Fragegruppe wird nur der Name angegeben, bei einer Frage die Frage. Die Elemente werden dann als Baum im Question-Bereich angezeigt. In dem Baum kann der Benutzer die Elemente verschieben bzw. zuordnen. Dabei können einem Ziel Fragen und Fragegruppen per Drag-and-Drop zugeordnet werden, wobei einer Fragegruppe nur Fragen zugeordnet werden können.

Ziele, Fragegruppen und Fragen besitzen weitere Eigenschaften, die über das Properties-Fenster in Eclipse verändert werden können. Sollte das Properties-Fenster nicht angezeigt werden, kann sich der Benutzer mit Rechtsklick auf eines der Elemente das Kontextmenü zum ausgewählten Element öffnen und über ⟨`Show Properties View`⟩ die Properties View anzeigen lassen.

Fragen haben folgende Eigenschaften:

- Defined Quality: Fragen können Qualitätsattribute zugeordnet werden, diese werden aber erst im Qualitätsmodell angelegt.
- Value: Die Frage selber.

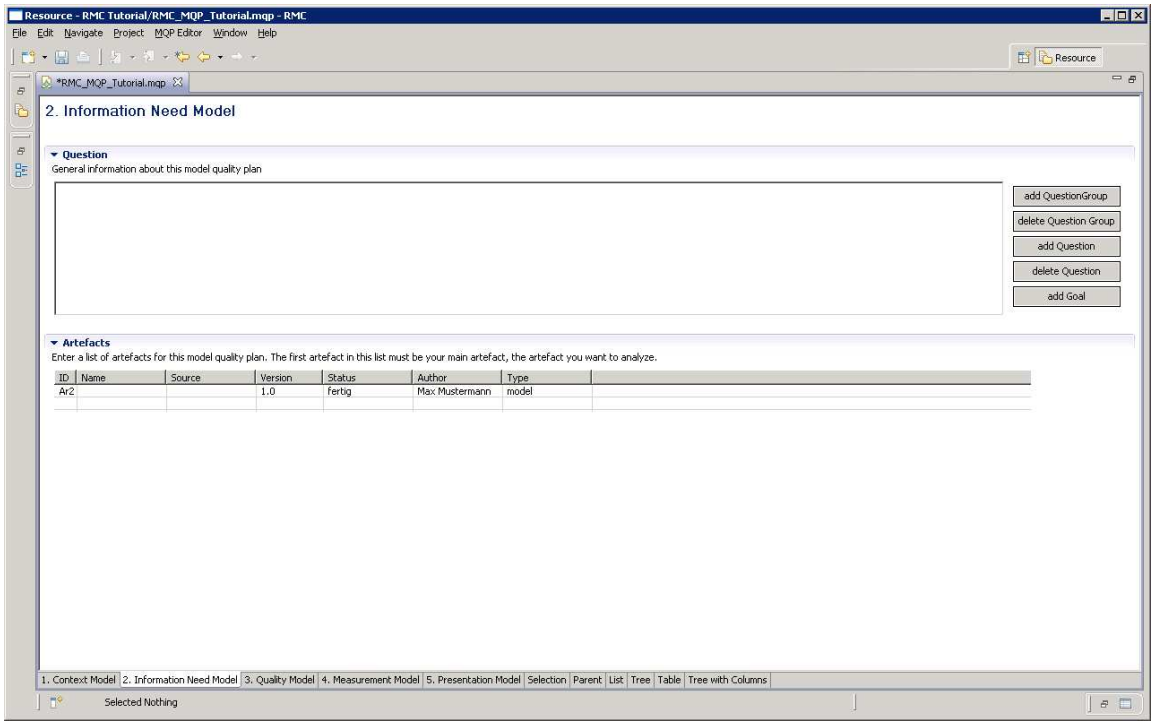

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
Resource
*RMC_MQP_Tutorial.mqp
2. Information Need Model
Question
General information about this model quality plan
add QuestionGroup
delete Question Group
add Question
delete Question
add Goal
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
Ar2 1.0 fertig Max Mustermann model
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns
Selected Nothing

Abbildung 4.28: MQP-Editor: Informationsbedürfnismodell

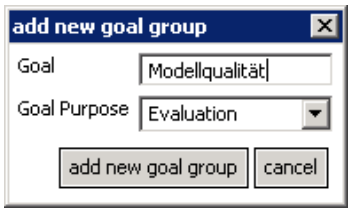


Abbildung 4.29: MQP-Informationsbedürfnismodell: Neues Ziel anlegen

Fragegruppen haben folgende Eigenschaften:

- Name: Name der Gruppe.

Ziele haben folgende Eigenschaften:

- Context Element: Der Kontext des Ziels.
- Name: Das Ziel.
- Object Of Study: Bezieht sich immer das Hauptartefakt, wird automatisch gesetzt.
- Purpose: Absicht des Ziels.
- Quality Focus: Ziele können Qualitätscharakteristiken zugeordnet werden, diese werden aber erst im Qualitätsmodell angelegt.
- View Point: Einem Ziel können Rollen aus dem Kontextmodell zugeordnet werden.

4.2.3.5 MQP-Qualitätsmodell

Als nächstes sollte vom Benutzer das Qualitätsmodell (Quality Model) ausgefüllt werden. Um vom Informationsbedürfnismodell ins Qualitätsmodell zu gelangen, muss der Benutzer den Reiter **Quality Model** auswählen. Die Abbildung 4.30 zeigt ein beispielhaft ausgefülltes Qualitätsmodell. Das Qualitätsmodell besteht aus zwei Bereichen, einer Zeichenfläche (oben), indem ein Graph aus **Qualitätsattributen (QA)** und **Qualitätscharakteristiken (QC)** aufgebaut werden kann, und einem Fragenbereich.

Im Fragenbereich (Question) werden nur die Ziele, Fragegruppen und Fragen dem Benutzer zur Unterstützung angezeigt. Im oberen Bereich steht dem Benutzer ein Canvas zur Verfügung. Rechts von der Canvas befindet sich eine Palette (siehe Abbildung 4.31) mit Elementen, die auf dem Canvas angezeigt werden können. Zur Verfügung stehen **Qualitätsattribute (QA)**, **Qualitätscharakteristiken (QC)** und Verbindungen (**Connections**). In die Elemente kann direkt der Name geschrieben werden. Aus den beiden Elementen lässt sich ein Graph aufbauen. Dafür gibt es das Verbindungsobjekt. Verbindungen kann der Benutzer zwischen Qualitätscharakteristiken untereinander und zwischen Qualitätscharakteristiken und Qualitätsattributen ziehen.

> **Hinweis:** Hier Reihenfolge beachten: Es ist nur möglich eine Verbindung von QC zu QA zu ziehen, nicht umgekehrt

QA und QC haben neben dem Namen noch weitere Eigenschaften. Zum Beispiel sollte der Benutzer dem QA Fragen und dem QC Ziele zuordnen. Dies kann über das Property-Fenster geschehen. Dort werden alle Eigenschaften der QA und QC angezeigt und können editiert werden.

QC haben folgende Eigenschaften:

- Definition
- Goal
- Model Quality

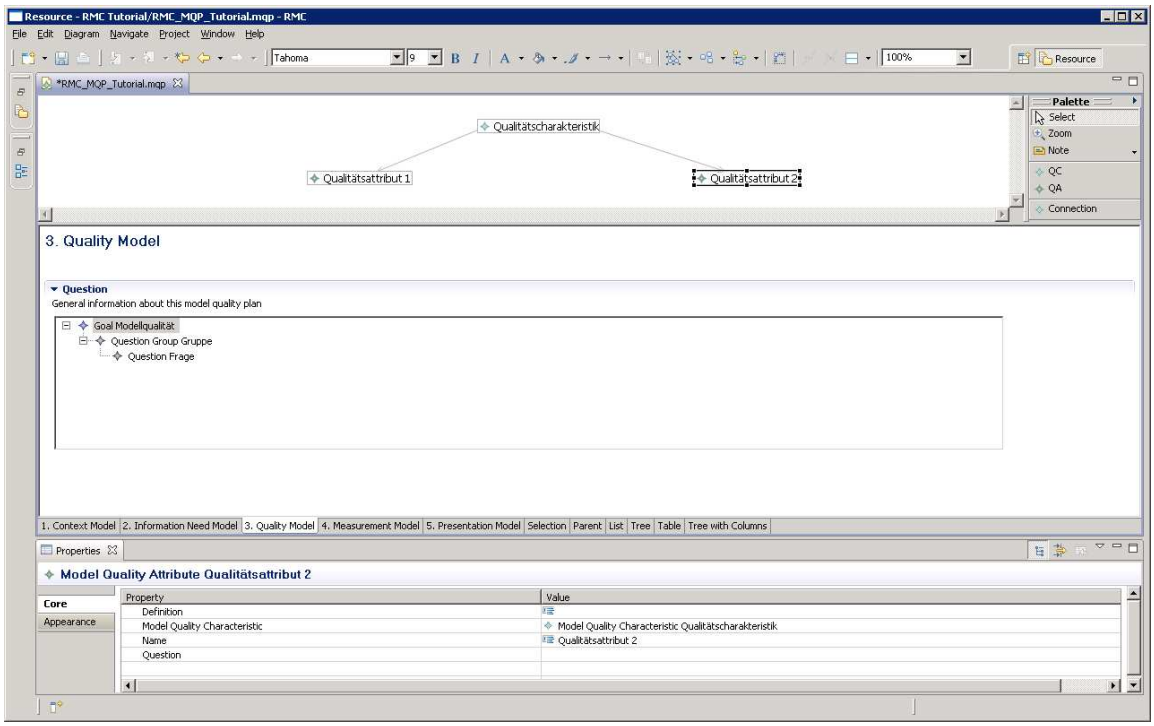


Abbildung 4.30: MQP-Editor: Qualitätsmodell

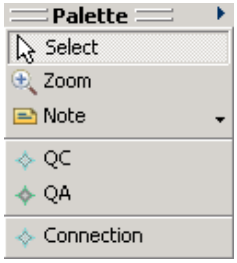


Abbildung 4.31: MQP-Qualitätsmodell: Palette der Zeichenfläche

- Model Quality Attribute
- Name
- Sub Characteristic
- Superior Characteristic

QA haben folgende Eigenschaften:

- Definition
- Model Quality Characteristic
- Name
- Question

4.2.3.6 MQP-Messmodell

Als nächstes sollte vom Benutzer das Messmodell (**Measurement Model**) eingegeben werden. Um vom Qualitätsmodell ins Messmodell zu gelangen, muss der Benutzer den Reiter **Measurement Model** auswählen. Die Abbildung 4.32 zeigt ein beispielhaft ausgefülltes Messmodell. Das Messmodell besteht aus zwei Bereichen. Im oberen Bereich können vom Benutzer den Qualitätsattributen (links), die zuvor im Qualitätsmodell angelegten wurden, vordefinerte Messungen (rechts) zugeordnet werden. Im unteren Bereich kann der Anwender ein Diagramm auswählen, auf dem dann die ausgewählten Messungen ausgeführt werden.

Eine detaillierte Definition der im System vordefinierten Messungen, mit ihrem jeweiligen OCL-Queries, befindet sich im Anhang unter B.

Im oberen Bereich stehen dem Benutzer zwei Bäume zur Verfügung. Der linke Baum enthält die im Qualitätsmodell angelegten Qualitätsattribute. Der rechte Baum enthält bereits vordefinierte Messungen und Indikatoren, die in einer Konfigurationsdatei (**mdef.mqp** im Projektordner) gespeichert sind. Die Messungen werden unterschieden nach Basismessungen (**Base Measures**) und abgeleitete Messungen (**Derived Measures**). Basismessungen sind Messungen die nur aus OCL und deren Funktionen bestehen. Abgeleitete Messungen können aus OCL und deren Funktionen sowie anderen Basis- oder abgeleiteten Messungen bestehen. Indikatoren (**Indicators**) bewerten bestimmte Eigenschaften anhand von Messergebnissen.

In der mit RMC ausgelieferten Konfigurationsdatei befinden sich schon einige vordefinierte Messungen und Indikatoren. Der Anwender kann zusätzlich noch weitere Messungen und Indikatoren über [add measure] und [add indicator] anlegen. In Abbildung 4.33 ist der Dialog zum Eingeben einer Messung dargestellt. Der Benutzer muss hier den Namen und das Akronym der Messung eingeben, sowie die Skala, die Einheit der Messung, und ob die Messung eine Basis- oder eine abgeleitete Messung ist, auswählen. Für die Messung selbst steht dem Benutzer ein mehrzeiliges Textfeld zur Verfügung. In diesem Feld kann der Benutzer die Messungen in OCL eingeben. Zur Unterstützung stehen dem Anwender einige OCL-Funktionen rechts in der Liste zur Verfügung. Diese Funktionen kann er per Drag-and-Drop in das Textfeld ziehen. Hat der Benutzer **derived** ausgewählt, werden ihm zusätzlich noch alle anderen bereits definierten Messungen in einer Liste angezeigt. Diese können auch

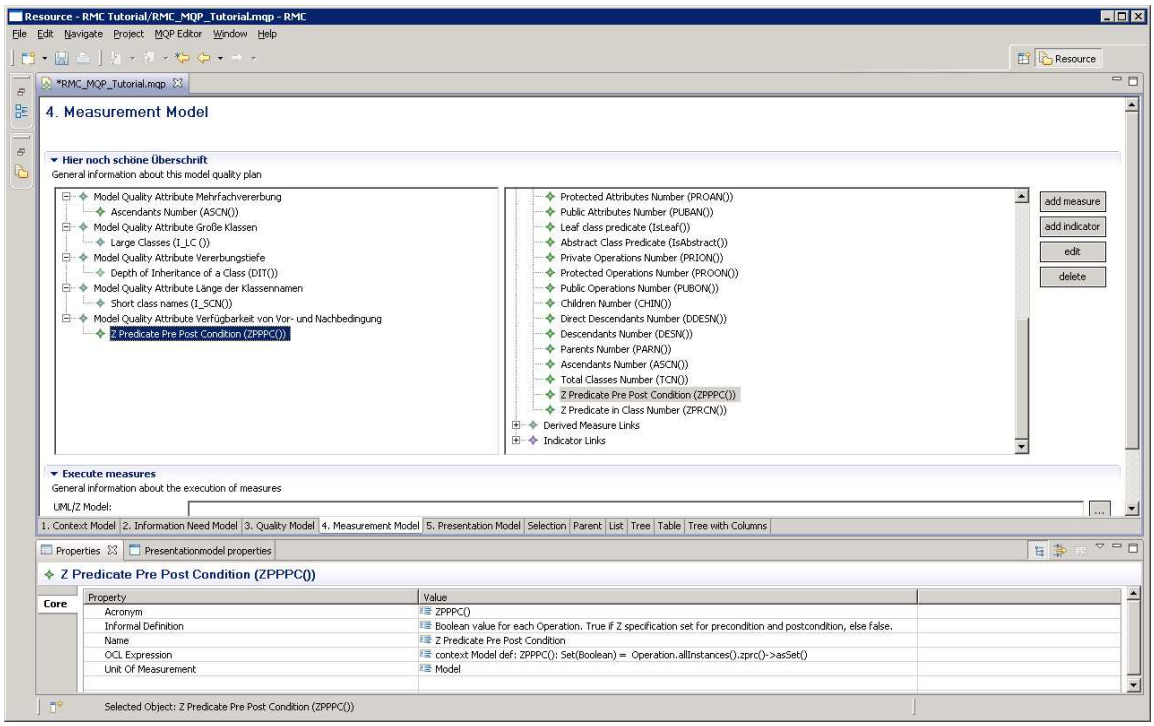

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
*RMC_MQP_Tutorial.mqp
4. Measurement Model
Hier noch schöne Überschrift
General information about this model quality plan
Model Quality Attribute Mehrfachvererbung
Ascendants Number (ASCN())
Model Quality Attribute Große Klassen
Large Classes (I_LC ())
Model Quality Attribute Vererbungstiefe
Depth of Inheritance of a Class (DIT())
Model Quality Attribute Länge der Klassennamen
Short class names (I_SCN())
Model Quality Attribute Verfügbarkeit von Vor- und Nachbedingung
Z Predicate Pre Post Condition (ZPPPC())
Protected Attributes Number (PROAN())
Public Attributes Number (PUBAN())
Leaf class predicate (IsLeaf())
Abstract Class Predicate (IsAbstract())
Private Operations Number (PRION())
Protected Operations Number (PROON())
Public Operations Number (PUBON())
Children Number (CHIN())
Direct Descendants Number (DDESN())
Descendants Number (DESN())
Parents Number (PARN())
Ascendants Number (ASCN())
Total Classes Number (TCN())
Z Predicate Pre Post Condition (ZPPPC())
Z Predicate in Class Number (ZPRCN())
Derived Measure Links
Indicator Links
add measure
add indicator
edit
delete
Execute measures
General information about the execution of measures
UML/Z Model:
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns
Properties
Presentationmodel properties
Z Predicate Pre Post Condition (ZPPPC())
Core
Property
Value
Acronym
ZPPPC()
Informal Definition
Boolean value for each Operation. True if Z specification set for precondition and postcondition, else false.
Name
Z Predicate Pre Post Condition
OCL Expression
context Model def: ZPPPC(): Set(Boolean) = Operation.allInstances().zprc()->asSet()
Unit Of Measurement
Model
Selected Object: Z Predicate Pre Post Condition (ZPPPC())

Abbildung 4.32: MQP-Editor: Messmodell

per Drag-and-Drop in das Textfeld für die Messung gezogen werden. Zusätzlich steht dem Benutzer noch ein mehrzeiliges Textfeld (**informal definition**) zur Verfügung, in dem er die Messung beschreiben kann. Dieses Feld ist optional. Um die Messung zu speichern, muss der Anwender [Save Changes] drücken. Dann wird die neu definierte Messung in der Konfigurationsdatei gespeichert und auch im rechten Baum im Messmodell angezeigt. Sie kann jetzt wie alle anderen Messungen einem QA per Drag-and-Drop zugewiesen werden.

In Abbildung 4.34 ist der Dialog zum Eingeben eines Indikators dargestellt. Der Benutzer muss im oberen Bereich den Namen und das Akronym der Messung eingeben, sowie die Skala und die Einheit der Messung auswählen. Zwischen der Skala und der Einheit der Messung befindet sich ein mehrzeiliges Textfeld (**informal definition**) für die Beschreibung des Indikators. Dieses Feld ist optional vom Benutzer auszufüllen. In der Mitte des Dialogs befindet sich eine Liste (**defined measures**) mit vordefinierten Messungen, die per Drag-and-Drop in die Textfelder zur Berechnung (**calculations**) weiter unten gezogen werden können. Unter den Messungen sind einige Operationen vorhanden, die auch, wie bereits bei den Messungen beschrieben, per Drag-and-Drop in die Berechnungsfelder gezogen werden können. Sie befinden sich ganz unten im Dialog. Durch die Berechnungsfelder kann der Benutzer bestimmte Eigenschaften bewerten, zum Beispiel die Anzahl von Methoden in einer Klasse. Auf der linken Seite befinden sich, die mit „IF“ bezeichneten Felder, auf der rechten Seite, die dazugehörigen „THEN“-Felder. Gibt der Benutzer eine Bedingung in ein „IF“-Feld ein (Anzahl Methoden kleiner drei), muss auch eine Bewertung in das dazugehörige „THEN“-Feld eingetragen werden (zum Beispiel eine Schulnote wie „zwei“). Für den „ELSE“-Fall, der nicht über die „IF“-Bedingungen abgedeckt ist, muss im

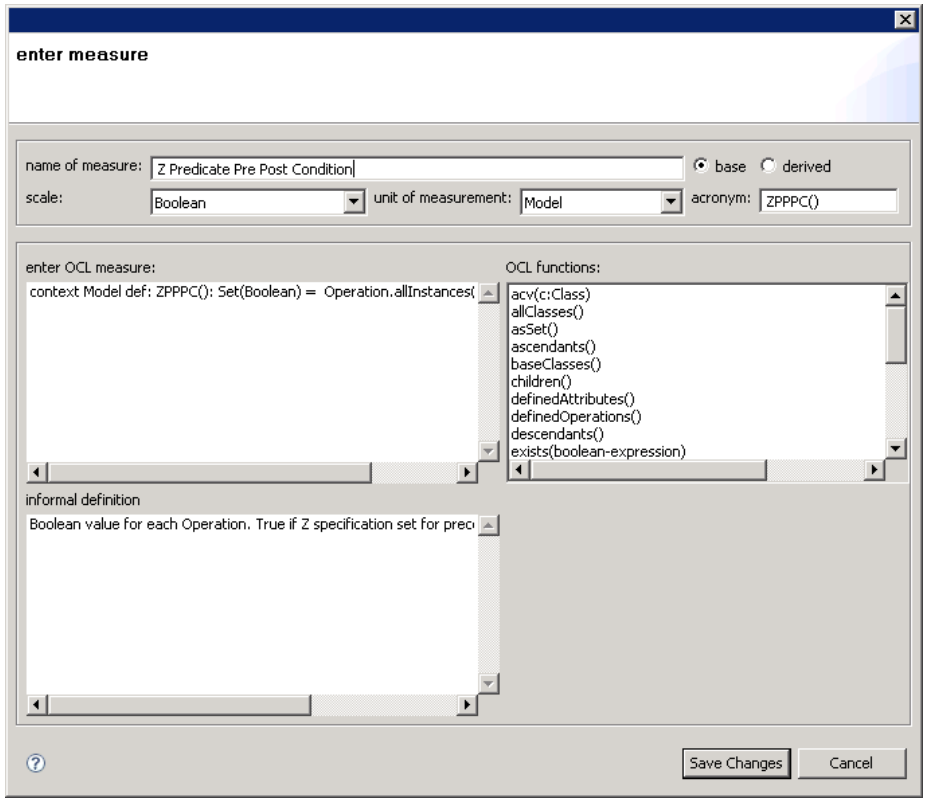


Abbildung 4.33: MQP-Messmodell: Neue Basismessung eingeben

letzten „Then“-Feld auch eine Bewertung eingefügt werden, bevor der Anwender den Indikator über [Save Changes] abspeichern kann. Der Indikator wird dann wie die Messungen in der Konfigurationsdatei abgespeichert und im Messmodell im rechten Baum dargestellt. Der Indikator kann jetzt wie alle anderen Messungen und Indikatoren auch einem QA per Drag-and-Drop zugewiesen werden.

Nachdem der Benutzer den QA-Messungen und Indikatoren zugewiesen hat, kann er diese Messungen auf einem UML/Z-Modell ausführen. Dafür gibt es im Messmodell unten den Bereich **execute measures**. Hier muss der Benutzer ein UML/Z-Modell auswählen. Dafür muss der Anwender neben dem Textfeld „UML/Z-Model“ auf [...] drücken, und ein neues Fenster zur Auswahl der UML/Z-Modell Datei öffnet sich.

> **Hinweis:** Das Modell muss sich dafür im Projektverzeichnis befinden.

Der Benutzer kann das passende Modell auswählen und über [Ok] bestätigen. Um dann die Messung auf dem Modell zu starten, muss der Anwender auf [do Measurement] drücken.

4.2.3.7 MQP-Präsentationsmodell

Das noch nicht vorgestellte Modell ist das Präsentationsmodell (**Presentation Model**). In diesem Modell werden dem Benutzer die Daten der vorherigen Modelle angezeigt (siehe Abbildung 4.35), unter anderem auch die Ergebnisse der Messungen aus dem Messmodell. Um vom Messmodell ins Präsentationsmodell zu gelangen, muss der Benutzer den Reiter **Presentation Model** auswählen.

Das Präsentationsmodell besteht aus zwei Bereichen. Links werden dem Benutzer die Eingaben der ersten vier Modelle als Graph visualisiert und rechts kann der Benutzer einstellen, wie und was visualisiert werden soll. Außerdem lässt sich hier die Diagnose anstoßen.

Bei der Visualisierung haben alle Elemente, die zum selben Teilmodell gehören, auch die gleiche Farbe.

- **Context Model:** grün
- **Information Need Model:** hellblau
- **Quality Model:** grau
- **Measurement Model:** blau

Standardmäßig werden alle Elemente aller Modelle angezeigt. Möchte der Benutzer nicht alle Elemente sehen, um zum Beispiel die Übersichtlichkeit zu erhöhen, kann er die Elemente bestimmter Modelle ausblenden. Für jedes Modell gibt es einen Hacken rechts im Einstellungsbereich. Entfernt der Benutzer einen Haken, werden die dazugehörigen Elemente des Modells nicht mehr angezeigt. Klickt der Benutzer mit der rechten Maustaste in das Visualisierungsfenster wird das komplette Modell auf die passende Größe skaliert, so dass der komplette Graph im Fenster angezeigt wird.

In der Visualisierung werden von den Elementen nur die Namen angezeigt. Möchte der Anwender noch die restlichen Eigenschaften der Elemente sehen, muss er ein Element

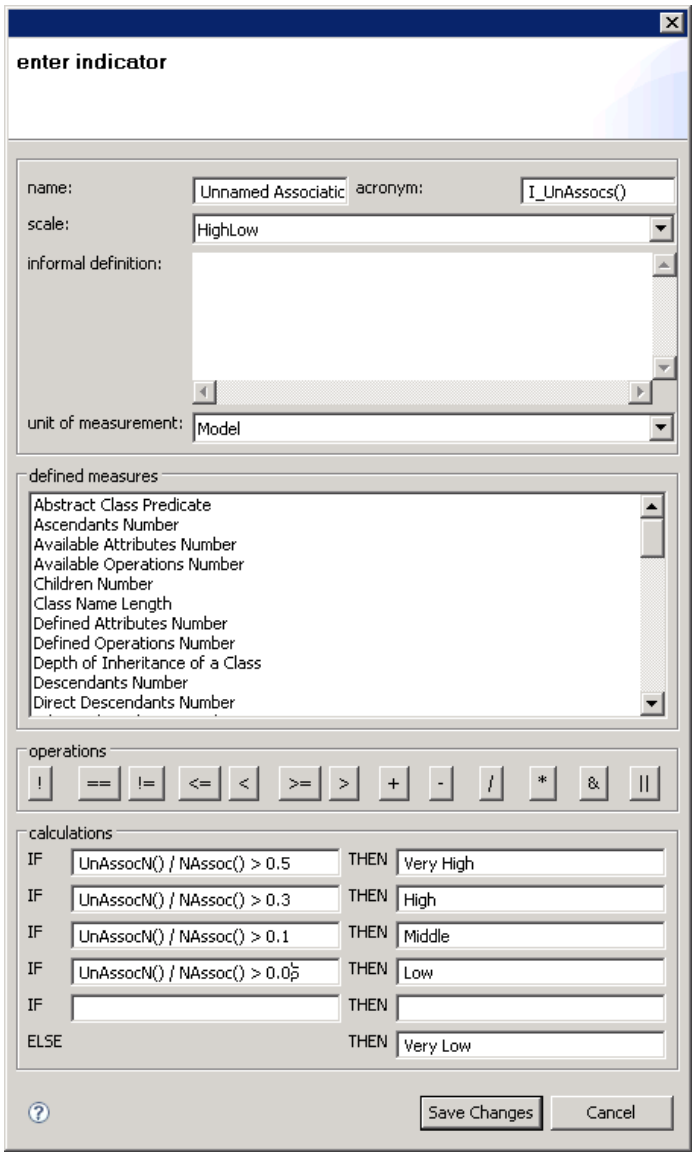


Abbildung 4.34: MQP-Messmodell: Neuen Indikator eingeben

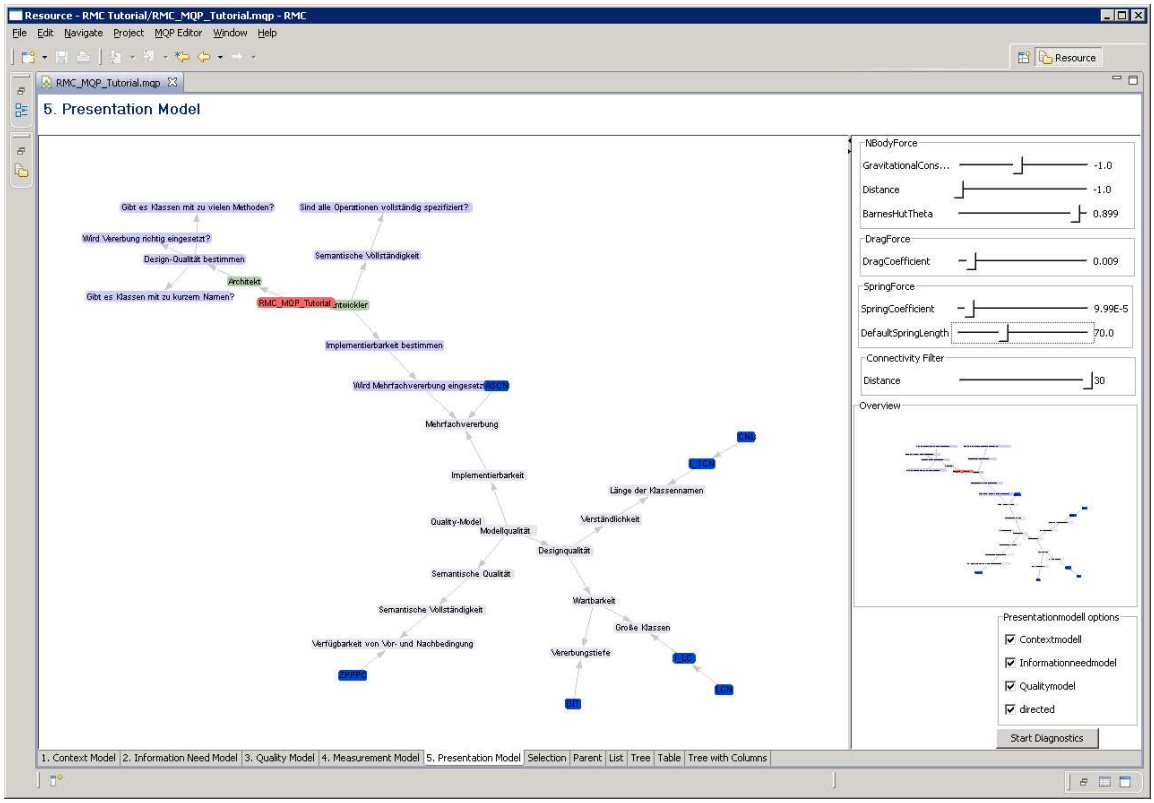


Abbildung 4.35: MQP-Präsentationsmodell

auswählen, und die zu dem ausgewählten Element gehörigen Eigenschaften werden in der Properties View von Eclipse angezeigt. Hier kann sich der Benutzer auch die Ergebnisse der Messungen aus dem Messmodell ansehen.

Um die Messergebnisse an die Diagnose zu übermitteln, muss der Benutzer [Start Diagnostics] drücken.

4.2.4 Diagnose

Im folgenden Kapitel wird die Bedienung der Diagnose beschrieben. Im Anschluss daran wird ein Überblick über die von der Diagnosekomponente verwendete Konfigurationsdatei und deren Struktur gegeben. Nähere Informationen über die Aufbau und die Erweiterung der Konfigurationsdatei kann unter Kapitel 7.2.4.1 nachgelesen werden.

4.2.4.1 Benutzung der Diagnose

Die Diagnose wird durch Anklicken des [Start Diagnostics]-Buttons im **Präsentationsmodell** gestartet. Voraussetzung für die Nutzung der Diagnose ist, dass eine Messung im **Messmodell** des MQPs durchgeführt wurde. Innerhalb der Diagnose können nur aus der Konfigurationsdatei (siehe Abschnitt 4.2.4.2) bekannte Indikatoren weiter verarbeitet werden, unbekannte werden nicht berücksichtigt.

In Abhängigkeit zum Grenzwert (zu sehen im **Point of view**-Abschnitt unter **Selected Value**in Abbildung 4.36) werden die Diagnosen und dazu passende Therapien dem Benutzer im Abschnitt **Diagnosis** angezeigt.

Dieser Grenzwert kann im Abschnitt **Point of View** herauf- bzw. herabgesetzt werden. Erhöhen des Wertes bedeutet, dass nur noch schwerwiegendere Modelldefekte bzw. Diagnosen und zugehörige Therapien angezeigt werden. Ein Herabsetzen des Wertes bedeutet: auch für nicht so gravierende Modelldefekte werden Diagnosen und passende Therapien anzeigt. Ist der Grenzwert null, so werden für alle der Diagnose bekannten Indikatoren Diagnosen und Therapien angezeigt.

Im Abschnitt **Diagnosis** (siehe Abbildung 4.36) werden in einer Baumstruktur Diagnosen und eine Ebene tiefer die dazugehörigen Therapien angezeigt. Klickt man auf eine Diagnose oder Therapie (gemeint ist hier der Name und nicht eine Checkbox) so wird, falls vorhanden im Abschnitt **Description** ein Beschreibungstext angezeigt. Handelt es sich bei einer Therapie um eine Texttherapie, so werden dem Benutzer in Textform Verbesserungsvorschläge mitgeteilt.

In der unteren rechten Ecke der Diagnose befindet sich ein Button mit der Bezeichnung [Refactoring]. Wählt man aus den vorgeschlagenen Therapien Refactoringtherapien aus (durch Anklicken und Aktivieren der Checkboxen), so wird die Refactoringkomponente angesprochen und für das Durchführen eines Refactorings werden alle notwendigen Daten übergeben.

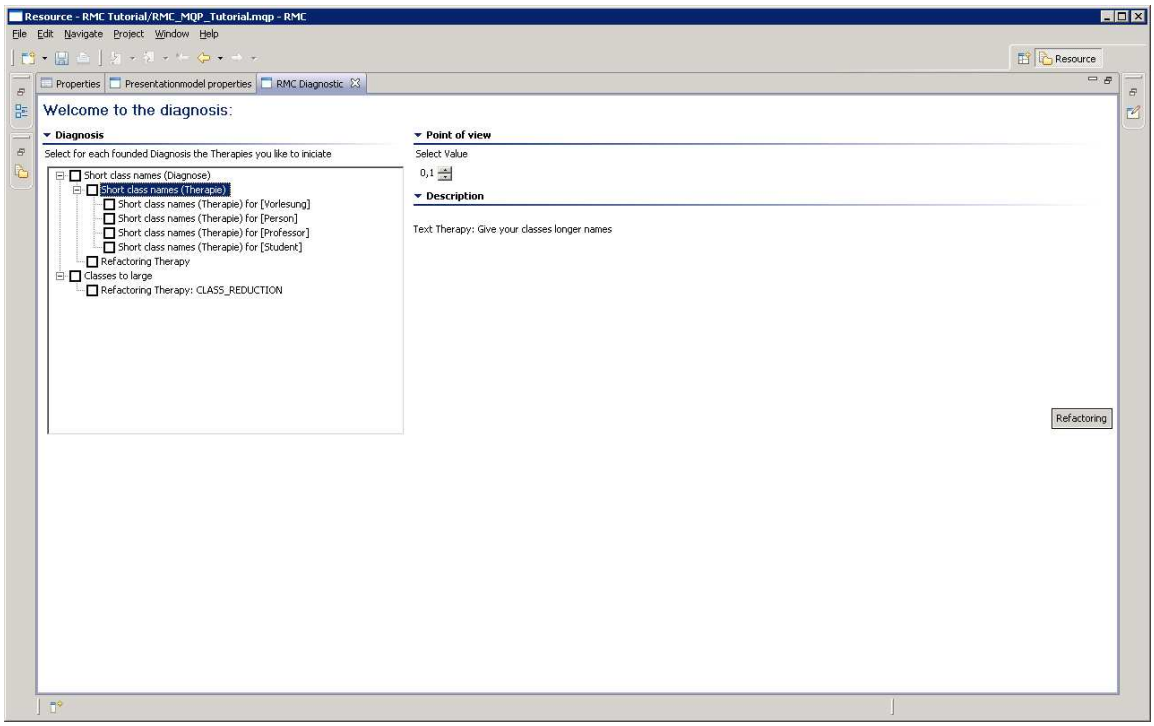


Abbildung 4.36: Diagnose: grafische Oberfläche

4.2.4.2 Konfigurationsdatei

Die Konfigurationsdatei der Diagnose liegt im XML Format vor. Sie befindet sich im **plug-ins** Ordner der Eclipse Installation im Unterordner **de.upb.pg.rmc.diagnostics**.

Im Folgenden wird nun auf die Struktur der Konfigurationsdatei näher eingegangen. Das Wurzelelement der Konfigurationsdatei heißt <configuration>. Darunter befinden sich Knoten für Parameter, Symptome, Diagnosen, Therapien und Normalisierungen. Im Folgenden soll nun auf diese Elemente näher eingegangen werden:

- <parameters>: Hier können Parameter eingetragen werden, welche in der kompletten Diagnosekomponente Verwendung finden sollen. Diese Parameter bestehen dabei aus einem Namen (**name**) und einem Wert (**value**):

```
<universalParameter name="..." value="..." />
```

Listing 4.1: Diagnoseknoten: <universalParameter>

- <symptoms>: Beschreibt alle der Diagnose derzeit bekannten Symptome <symptom>. Ein Symptom besteht dabei aus einem Namen, welcher weiterführend als Schlüssel für die weiter unten definierten Diagnosen verwendet wird. Des Weiteren besitzt jedes Symptom eine Normalisierung und genau einen zugewiesenen Indikator.

```
<symptom name="..." normalization="..." >
  <indicator name="..." />
</symptom>
```

Listing 4.2: Diagnoseknoten: <symptom>

> **Hinweis:** Der Name des Indikators wird zur Identifikation der Indikatoren verwendet, welche durch die Modelqualitätsplankomponente übergeben werden. Es ist also darauf zu achten, dass die Namen gleich sind.

- <diagnostics>: Umfasst alle der Diagnosekomponente bekannten Diagnosen. Die Namen der Symptome werden als Schlüssel verwendet und sind den Diagnosen als Referenzen (<reference name = "...">) zugewiesen. Jede Diagnose besteht zusätzlich zur Referenz aus einem Namen und aus einer Beschreibung. Der Name wird wiederum als Schlüssel für im Folgenden definierte Therapien verwendet. Die Beschreibung wird dem Benutzer bei Selektion einer Diagnose als Beschreibung der Diagnose angezeigt.

```
<diagnosis name="..." description="..." >
    <reference name="..." />
</diagnosis>
```

Listing 4.3: Diagnoseknoten: <diagnosis>

- <therapies>: Es werden zwei Arten von Therapien von RMC unterstützt, zum einen die Texttherapie und zum anderen Refactoringtherapien. Je nachdem, um welche Therapie es sich handelt, wird der Typ einer Therapie mit **text** für eine Texttherapie und **refactoring** für eine Refactoringtherapie gekennzeichnet. Weiter haben Therapien eine Referenz auf eine Diagnose und einen Beschreibungstext. Refactoringtherapien haben neben einer Referenz und einer Beschreibung zusätzlich noch eine Eigenschaft (siehe Eintrag **PropertyIdentifierList** in Tabelle 4.7 auf Seite 84), welche für das Erfragen nach eigentlichen Refactorings an die Refactoringkomponente genutzt werden.

```
<therapy name="..." type="..." >
    <property name="..." /> (nur bei Refaktoring Therapie)
    <reference name="..." />
    <description description="..." />
</therapy>
```

Listing 4.4: Diagnoseknoten: <therapy>

- <normalizations>: Für Normalisierungen müssen ein Typ sowie ein Name angegeben werden. Abhängig vom Normalisierungstyp kann die Normalisierung beliebig viele Parameter enthalten. Diese bestehen aus einem Namen und einem Wert.

```
<normalisation type="..." name="..." >
    <parameter name="..." value="..." />
  ...
</normalisation>
```

Listing 4.5: Diagnoseknoten: <normalisation>

Hinweis: Namen einzelner Teile innerhalb der Konfigurationsdatei werden häufig als Referenzen auf anderen Stellen verwendet. Dabei ist auf das Einhalten gleicher Namen zu achten.

4.2.5 Refactoring

In den folgenden beiden Abschnitten wird beschrieben, wie Refactorings gestartet und mit Unterstützung von Benutzereingaben durchgeführt werden können. Außerdem geht der Abschnitt 4.2.5.2 auf Optionen bei der Konfiguration von Refactorings ein.

4.2.5.1 Ausführen von Refactorings

Es besteht die Möglichkeit, Refactorings aus den folgenden Umgebungen zu starten:

- Klasseneditor
- Modelleditor
- Diagnose-View

Dieser Abschnitt geht ausschließlich auf die ersten beiden Optionen ein. Das Starten eines Refactorings aus der Diagnose wird im Abschnitt 4.2.4.1 im Detail beschrieben.

Das kontextsensitive ⟨`Refactoring`⟩-Menü befindet sich im Hauptmenü. Abhängig von den selektierten Elementen in der Modell- bzw. Klassendiagrammansicht werden ausführbare Refactorings angeboten. Refactorings, die nicht ausgeführt werden können, weil die Anwendbarkeitsbedingung (siehe Abschnitt 4.2.5.2, Seite 82) auf selektierten Elementen nicht erfüllt ist, werden ausgegraut dargestellt.

Einzelne Refactorings lassen sich konfigurieren. Darauf geht der Abschnitt 4.2.5.2 ab Seite 82 im Detail ein. Den einzigen Eintrag, der fest ins Menü integriert ist, findet man unterhalb der Trennlinie des Refactoringmenüs (Abbildung 4.37): ⟨`Reload`⟩. Wird eine Konfigurationsdatei nicht erfolgreich geparst bzw. sind keine Refactorings definiert, so wird nur der ⟨`Reload`⟩-Menüeintrag zur Auswahl angeboten.

Wurden Änderungen an der Konfigurationsdatei vorgenommen, so kann diese mit ⟨`Reload`⟩ neu geladen werden. Beim nächsten Menüaufruf werden die ⟨`Refactoring`⟩-Menüeinträge aktualisiert.

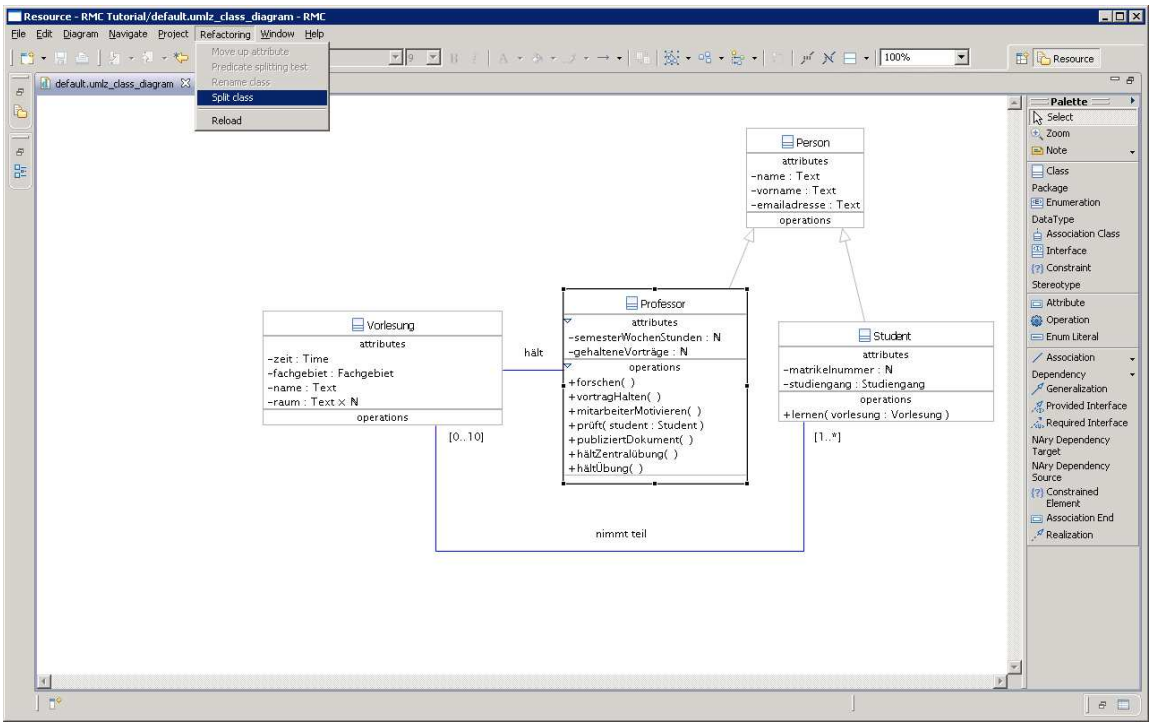


Abbildung 4.37: Klassendiagrammeditor: Refactoringmenü

Kann eine Konfigurationsdatei nicht eingelesen werden, erscheint eine Fehlermeldung, die die Fehlerposition in der zugrundeliegenden Datei angibt (Abbildung 4.38). Sobald der Fehler behoben ist, kann mit ⟨`Reload`⟩ die Konfigurationsdatei neu eingelesen werden.

Wird ein ausführbares Refactoring im Menü angewählt, erscheint der Refactoringassistent (Abbildung 4.39). Die Startseite des Wizards beschreibt die Änderungen, die das dazugehörige Refactoring im Modell bewirkt.

Mit der Betätigung des [Next]-Buttons von der Startseite des Wizards leitet man das Refactoring ein. Bevor Änderungen am Modell vom Refactoringablauf vorgenommen werden können, müssen in einigen Refactorings entsprechend der Definition aus der Konfigurationsdatei Entscheidungen getroffen werden. Drei Entscheidungsarten sind definiert:

- Namensgebung: Der Benutzer wird aufgefordert, eine Zeichenfolge vorzugeben. (Abbildung 4.40)
- Es wird eine Auswahl eines oder mehrerer Elemente aus einer Elementenmenge getroffen: Der Anwender selektiert eine Elementenmenge, die für das weitere Vorgehen eines Refactorings benötigt wird, indem die Checkboxen einzelner Zeilen der vorgegebenen Tabelle markiert werden. (Abbildung 4.41)
- Trennung von einem Prädikat in zwei Teilprädikate: Der Anwender bestimmt zwei Teilprädikate, deren Konjunktion semantisch mit dem vorgegebenen alten Prädikat übereinstimmt. Um die Übereinstimmung zu gewährleisten, wird intern ein Modelchecker verwendet. (Abbildung 4.42)

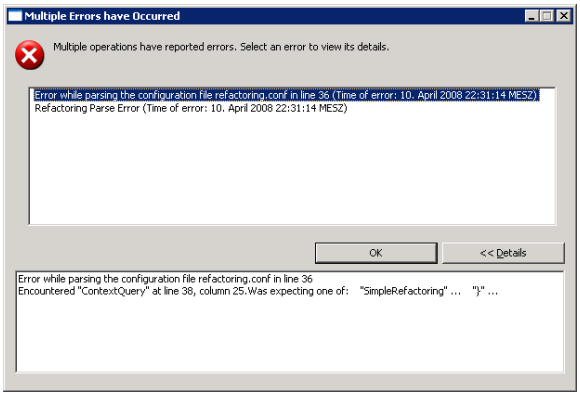

Multiple Errors have Occurred
Multiple operations have reported errors. Select an error to view its details.
Error while parsing the configuration file refactoring.conf in line 36 (Time of error: 10. April 2008 22:31:14 MESZ)
Refactoring Parse Error (Time of error: 10. April 2008 22:31:14 MESZ)
OK
<< Details
Error while parsing the configuration file refactoring.conf in line 36
Encountered "ContextQuery" at line 38, column 25.Was expecting one of: "SimpleRefactoring" ... "}" ...

Abbildung 4.38: Klassendiagrammeditor: Fehlermeldung beim Aufruf des Refactoring-menüs

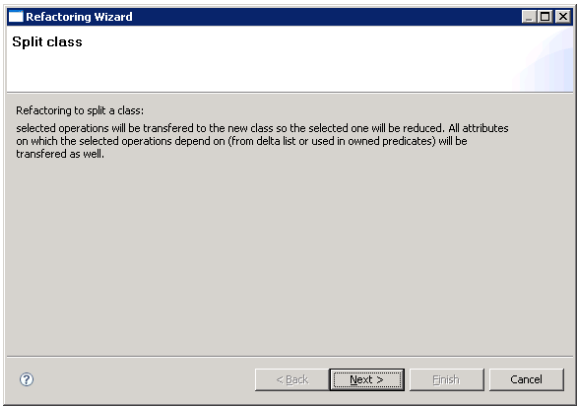

Refactoring Wizard
Split class
Refactoring to split a class:
selected operations will be transfered to the new class so the selected one will be reduced. All attributes on which the selected operations depend on (from delta list or used in owned predicates) will be transfered as well.
< Back
Next >
Finish
Cancel

Abbildung 4.39: Refactoringassistent: Startseite für Refactoring **Split Class**

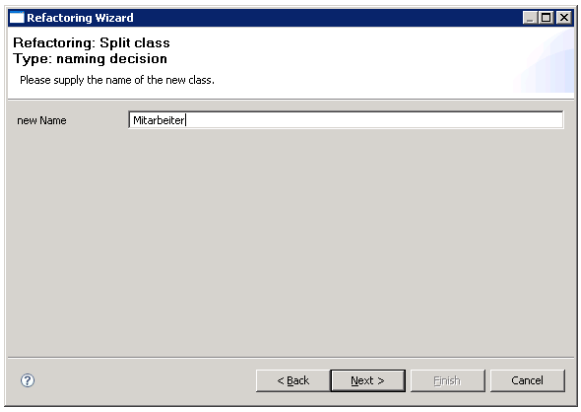


Abbildung 4.40: Refactoringassistent: Entscheidungsseite zur Angabe einer Zeichenfolge

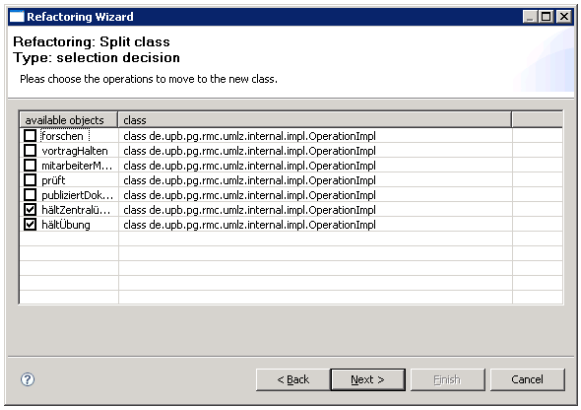


Abbildung 4.41: Refactoringassistent: Entscheidungsseite zur Auswahl eines oder mehrerer Elemente aus einer Elementenmenge beim Ausführen eines Refactorings

Die Art der Entscheidung, der Name des ausgewählten Refactorings sowie eine Beschreibung der aktuellen Entscheidung werden jeweils im oberen Teil der Wizard-Seiten eingeblendet (4.42). Entsprechen Angaben des Anwenders nicht definierten Bedingungen aus der Konfigurationsdatei, so erscheint eine Fehlermeldung im oberen Teil des Refactoringassistenten, die eine Fortführung des Refactorings verhindert (Abbildung 4.41). Der Status der Durchführung ist der letzten Seite des Refactoringassistenten zu entnehmen. Das Refactoring kann nachfolgend mit [Finish] abgeschlossen werden (Abbildung 4.43).

Ein Refactoring kann vollständig mit der **Undo**-Funktion des jeweiligen Editors zurückgenommen werden.

4.2.5.2 Konfigurationsdatei

Um eine einfache Handhabung beim Hinzufügen neuer Refactorings zu erreichen, wird eine Konfigurationsdatei für die Refactorings genutzt. Des Weiteren wurde zum Beschreiben der Refactorings eine Grammatik nach Backus-Naur-Form (BNF) entworfen. Dieses Kapitel

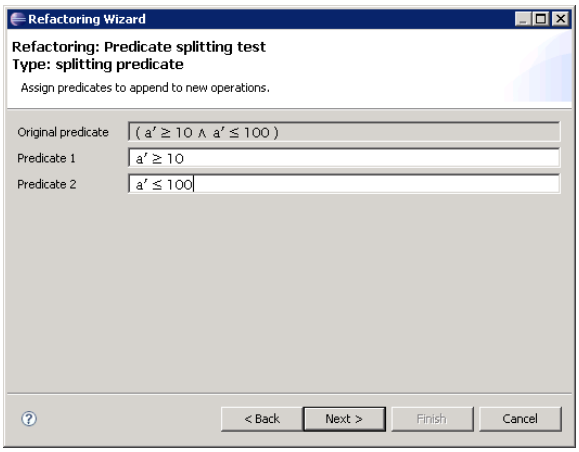


Abbildung 4.42: Refactoringassistent: Entscheidungsseite zur Auswahl von Teilprädikaten zu einem vorgegebenen Prädikat

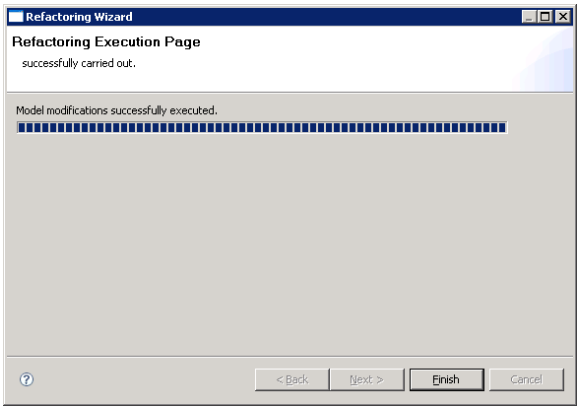


Abbildung 4.43: Refactoringassistent: Erfolgreicher Abschluss eines Refactorings

beschäftigt sich daher sowohl mit dem Aufbau der Grammatik als auch mit dem Aufbau der Konfigurationsdatei.

Von RMC einzulesende Refactorings können in einer der folgenden Dateien definiert sein:

1. im Eclipse-Workspace im Unterverzeichnis
   `.metadata/.plugins/de.upb.pg.rmc.refactoring/refactoring.conf`
2. im Eclipse-Installationsverzeichnis im Unterverzeichnis
   `/plugins/de.upb.pg.rmc.refactoring/refactoring.conf`
3. von RMC mitgelieferte Konfigurationsdatei

Es wird in der oben aufgeführten Reihenfolge nach einer Konfigurationsdatei gesucht. Sobald die Verfügbarkeit einer Datei gegeben ist, wird das Vorhandensein weiterer Dateien nicht überprüft. Mit der Definition von Refactorings in 1 können diese abhängig vom Arbeitsverzeichnis des Anwenders definiert und konfiguriert werden. Falls die Dateien 1 und 2 nicht vorhanden sind, bietet RMC bei Bedarf vordefinierte Refactorings an.

Die Grundlage für das Beschreiben von Refactorings stellt die in diesem Kapitel vorgestellte Grammatik dar. In der folgenden Tabelle sind alle Nichtterminale der Grammatik aufgeführt, die wichtigsten Einträge werden danach näher erläutert.

Tabelle 4.7: Refactoring: Nichtterminale der Grammatik

Nichtterminal	Produktionen	Beschreibung
CompilationUnit	Header PropertyIdentifierList (GlobalOcl)? TypeIdentifierList SimpleRefactorings ComplexRefactorings RefactoringTypes	Eine **CompilationUnit** ist das Startelement der Grammatik.
PropertyIdentifier-List	„PropertyList“ {(PropertyIdentifier)* }	In der **PropertyIdentifierList** werden alle möglichen Auswirkungen einzelner Refactorings angegeben, z.B. wirkt sich die Property CLASS_REDUCTION so aus, dass eine Klasse vom Umfang der Klassenelemente her verkleinert wird. In der Konfigurationsdatei der Diagnose (siehe Kapitel 4.2.4.1) werden die Eigenschaften ebenfalls aufgeführt. Damit ist die Diagnose-Komponente in der Lage ein Refactoring anzubieten.
TypeIdentifierList	„TypeIdentifierList“ {(TypeIdentifier)* }	Die **TypeIdentifierList** verfügt über eine Auflistung aller Datentypen, die in Refactoringdefinitionen dieser Konfigurationsdatei verwendet werden dürfen. Eine Verwendung von undefinierten Typen führt zu Fehlermeldungen.

Nichtterminal	Produktionen	Beschreibung
Header	„Header“ { „Refactoring grammar file of“ Product „Version“ < *VERSION_NR* > }	Der **Header** enthält den Namen und die Version der Refactoring-Komponente.
SimpleRefactorings	„SimpleRefactorings“ { (SimpleRefactoring)* }	In **SimpleRefactorings** werden einfache Refactorings (**SimpleRefactoring**) beschrieben.
ComplexRefactorings	„ComplexRefactorings“ { (ComplexRefactoring)* }	In **ComplexRefactorings** werden komplexe Refactorings (**ComplexRefactoring**) beschrieben.
RefactoringTypes	„RefactoringTypes“ { (RefactoringType)* }	In **RefactoringTypes** werden die angebotenen Refactoringtypen (**RefactoringType**) beschrieben.
Product	„RMC-Plugin for Eclipse“	Name des entwickelten Tools.
GlobalOcl	„GlobalOCL“ {< *STRING_LITERAL_BESSER* > }	Hierin werden globale OCL-Funktionen definiert.
RefactoringType	„RefactoringType“ { RefactoringTypeName RefactoringTypeIdentifier Refactoring Properties TypeDescription ApplicabilityCondition StumblingBlock Decisions }	Die Definition eines Refactoringtypen.
Refactoring	„Refactoring“ < *IDENTIFIER* >	Dies ist der eindeutige Name eines **Refactorings**, das in einem der Abschnitte **SimpleRefactorings** bzw. **ComplexRefactorings** definiert ist.
RefactoringType-Name	„Name“ StringLiteral	Name des Refactoringtypen, der u.a. im Refactoringmenü eingeblendet wird.
TypeDescription	„Description“ StringLiteral	Dies ist die Beschreibung des Refactoringtypen, wird im **Refactoring-Wizard** angezeigt.
Properties	„Properties“ { (PropertyIdentifier)* }	Hier sind die Auswirkungen eines Refactorings beschrieben.
Applicability-Condition	„ApplicabilityCondition“ { (ContextQuery)* }	Die **ApplicabilityCondition** ist eine Prüfung, welche Refactorings auf den selektierten Modellelementen ausführbar sind.
StumblingBlock	„StumblingBlock“ { (ContextQuery)* }	Beschreibt erste Zuweisungen für das Ausführen eines Refactorings.

Nichtterminal	Produktionen	Beschreibung
Decisions	„Decisions“ { (Decision)* }	Die Definition von Entscheidungen.
VariableType	< *IDENTIFIER* > („.“ < *IDENTIFIER* >)*	Gibt vollqualifizierte Namen der Typen wieder z.B. `de.upb.rmc.umlz.NamedElement`.
Decision	„Decision“ { DecisionType (ContextQuery)? DecisionDescription DecisionReturnVariable (DecisionValidation)+ }	Beschreibt eine Entscheidung, die vom Anwender zu treffen ist.
DecisionType	„Type“ DecisionTypeElement	Der Typ einer Entscheidung.
DecisionType-Element	„StringInputDecision“ | „ListQueryDecision“ | „PredicateSplitDecision“	Ein **DecisionTypeElement** ist entweder eine **StringInputDecision** oder **ListQueryDecision** oder **PredicateSplitDecision**. Eine **StringInputDecision** ist eine Entscheidung, die eine Eingabe einer Zeichenfolge zur Folge hat. Bei einer **ListQueryDecision** wird ein Element aus einer Liste ausgewählt und bei einer **PredicateSplitDecision** wird ein Prädikat in zwei Bestandteile aufgeteilt.
DecisionDescription	„Description“ StringLiteral	Die Beschreibung einer Entscheidung.
DecisionValidation	„DecisionValidation“ { DecisionValidationDescription DecisionValidationQueries ContextQueryList }	Eine Bedingung deren Validierung angibt, ob das Refactoring mit der vom Anwender getroffenen Entscheidung durchführbar ist. Nach jeder Änderung im Entscheidungsdialog wird diese neu ausgeführt.
DecisionValidation-Description	„DecisionValidationDescription“ StringLiteral	Hierin wird der Grund für das Misslingen der Validierung der `DecisionValidation` angegeben, die auch im Wizard angezeigt wird.
DecisionReturn-Variable	„DecisionReturnVariable“ { VariableIdentifier }	In dieser Variable wird das Ergebnis der Entscheidung gespeichert.
Variable	„Variable“ { VariableType VariableName }	Die Definition einer Variablen.
VariableName	< *IDENTIFIER* >	Name der Variablen.
SimpleRefactoring	„SimpleRefactoring“ { RefactoringBase AfterTemplate }	**SimpleRefacoring** beschreibt die Definition eines einfachen Refactorings.

Nichtterminal	Produktionen	Beschreibung
ComplexRefactoring	„ComplexRefactoring“ { RefactoringBase SubRefactorings }	**ComplexRefacoring** beschreibt die Definition eines komplexen Refactorings.
RefactoringBase	„RefactoringName“ RefactoringParameter BeforeTemplate Precondition (EnforcedRefactorings)? Calculation	**RefactoringBase** beschreibt die Bestandteil, die sowohl ein einfaches als auch ein komplexes Refactoring enthalten können.
Refactoring- Parameter	„Parameter“ { (VariableType (VariableName)* „;“)* }	Hier sind die Variablen definiert, auf welchen das Refactoring arbeitet.
RefactoringName	„Name“ {< *IDENTIFIER* >}	Der Name des Refactorings.
BeforeTemplate	„BeforeTemplate“ ContextQueryList	Definition nach [RW07]. Hier werden Variablen mit Modellelementen vorbelegt.
Calculation	„Calculation“ ContextQueryList	Definition nach [RW07]. Hier findet eine Berechnung von Variablenwerten statt.
Precondition	„Precondition“ ContextQueryList	Definition nach [RW07]. In einer **Precondition** werden einer oder mehrere gegebene Parameter auf Vorbedingungen überprüft. Nur wenn diese Bedingung erfüllt sind, kann das Refactoring fortgeführt werden.
AfterTemplate	„AfterTemplate“ ContextQueryList	Definition nach [RW07]. In einem **AfterTemplate** sind die durchzuführenden Änderungen beschrieben. Ein **AfterTemplate** ist nur Bestandteil eines einfachen Refactorings.
SubRefactorings	„SubRefactorings“ { (SubRefactoring)+ }	**SubRefactorings** definiert eine Liste von Subrefactorings, die in einem komplexen Refactoring durchgeführt werden.
SubRefactoring	„SubRefactoring“ { SubRefactoringName { RefactoringIdentifier } SubRefactoringCondition VariableAssignments SubQueries ContextQueryList SubCalculation ContextQueryList }	Ein **SubRefactoring** definiert Ausführungsbedingungen und Zuweisungen eines durchzuführenden Subrefactorings.

Nichtterminal	Produktionen	Beschreibung
SubRefactoring-Condition	„SubRefactoringCondition“ ContextQueryList	Bedingung für ein **SubRefactoring**. Solange die hier definierte Bedingung erfüllt ist, wird ein in `SubRefactoring` angegebenes Refactoring wiederholt ausgeführt.
EnforcedRefactorings	„EnforcedRefactorings“ { (EnforcedRefactoring)+ }	Gibt eine Menge zu erzwingender Refactorings an, die vor der Ausführung des `AfterTemplates` bzw. der `SubRefactorings` ausgeführt werden könnten, falls die `Precondition` nicht erfolgreich validiert wird.
EnforcedRefactoring	„EnforcedRefactoring“ { RefactoringName { RefactoringIdentifier } VariableAssignments EnforcedQueries ContextQueryList }	Die Beschreibung eines zu erzwingenden Refactorings.
VariableAssignments	„VariableAssignments“ { (VariableAssignment)* }	Enthält eine Liste von Variablenzuweisungen.
VariableAssignment	„VariableIdentifier“ = VariableIdentifier „;“	Beschreibt eine Variablenzuweisung, wobei der linke Eintrag von einem Unterrefactoring stammt und der rechte Eintrag von einem Oberrefactoring.
ContextQueryList	{ (ContextQuery)* }	Beschreibt eine Liste von OCL-Ausdrücken.
ContextQuery	„ContextQuery“ { (Variable)? Parameter Query }	Beschreibung einer OCL-Anfrage, gibt entweder eine Boolsche Variable oder eine Variable, die im Typ angegeben ist zurück.
Parameter	„Parameter“ { (VariableType (VariableName)* „;“)* }	Variablen, die in einer OCL-Anfrage verwendet werden.
Query	„Query“ StringLiteral	Eine OCL-Anfrage.
RefactoringIdentifier	< *IDENTIFIER* >	Eine Referenz auf das Refactoring (über den Refactoringnamen).
RefactoringType-Identifier	„Type“ < *IDENTIFIER* >	Eindeutiger Name eines Refactoringtyps.
VariableIdentifier	< *IDENTIFIER* >	Name einer Variablen.
PropertyIdentifier	< *IDENTIFIER* >	Name einer Eigenschaft.
TypeIdentifier	< *IDENTIFIER* > („.“ < *IDENTIFIER* >)*	Gibt vollqualifizierte Namen der Type wieder z.B. „de.upb.rmc.umlz.NamedElement“.
StringLiteral	< *STRING_LITERAL* >	Eine Zeichenfolge.

4.3 FAQ

4.3.1 Allgemeine Probleme

? Frage: Welche Java Version wird benötigt?

! Antwort: Es wird Sun Java 6 benötigt.

4.3.2 Projekte

? Frage: RMC wurde neu installiert. Sind die alten Projekte weiter nutzbar?

! Antwort: Bestehende Projekte können in RMC importiert werden: ⟨`File`⟩→⟨`Import`⟩→⟨`Existing Projects into Workspace`⟩. Die so ausgewählten Projekte erscheinen nun in RMC.

4.3.3 Editor

? Frage: Wie lege ich ein neues Diagram mit einem dazugehörigen neuen Modell an?

! Antwort: Das Anlegen eines Diagrammes inklusive eines neuen Modells ist in Kapitel 4.2.2.1 auf Seite 37 beschrieben.

? Frage: Wie lege ich ein Diagram zu einem bestehenden Modell an?

! Antwort: Das Anlegen eines Diagrammes auf einem bereits bestehenden Modell ist in Kapitel 4.2.2.1 auf Seite 37 beschrieben.

? Frage: Wie können Kardinalitäten von Assoziationen eingegeben werden?

! Antwort: Das Bearbeiten von Kardinalitäten ist in Kapitel 4.2.2.2 auf Seite 46 erläutert.

? Frage: Welche Default-Kardinalitäten werden im Klassendiagramm angenommen?

! Antwort: Für Kardinalitäten, die nicht explizit angelegt bzw. nicht im Diagramm angezeigt werden, gilt der Wert „1“.

4.3.4 Modellqualitätsplan

? Frage: Wie können Verbindungen (**Connections**) im Qualitätsmodell gelöscht werden?

! Antwort: Die Zuordnung in den Eigenschaften des Qualitätsattributs oder der Qualitätscharakteristik muss gelöscht werden. Um die Diagrammdarstellung zu aktualisieren, muss der MQP anschließend erst gespeichert werden.

4.3.5 Diagnose

? Frage: Im **Diagnosis** Abschnitt werden keine Diagnosen angezeigt.

! Antwort: Zwei Möglichkeiten sind vorhanden. Bei der ersten sind die Indikatoren der Diagnose noch nicht bekannt (Siehe: Erweitern der Diagnose-Konfigurationsdatei (Siehe Abschnitt 4.2.4.2). Bei der zweiten ist der Grenzwert **Point of View** wie in Abbildung 4.36 so hoch eingestellt, dass keine Modelldefekte angezeigt werden

4.3.6 Refactoring

? Frage: Warum stehen im ⟨`Refactoring`⟩-Menü keine Refactorings? Außer einem Eintrag zum erneuten Laden der Konfigurationsdatei ist das Menü leer.

! Antwort: Entweder enthält die Konfigurationsdatei keine Refactoringtypen oder die Konfigurationsdatei ist defekt. Nach einem erneuten Laden der Konfigurationsdatei sollte eine Fehlermeldung erscheinen, die über die fehlerhafte Stelle in der Konfigurationsdatei informiert. Erscheint sie nicht, ist die Datei korrekt formuliert und es sind keine Refactoringtypen definiert.

? Frage: Warum sind alle Refactorings im ⟨`Refactoring`⟩-Menü ausgegraut? Sind diese falsch definiert?

! Antwort: Einzelne Refactorings lassen sich nur ausführen, wenn bestimmte Modellelemente markiert sind. Zum Umbenennen einer Klasse verlangt die ausgelieferte Standardkonfiguration, dass *genau* eine Klasse markiert wurde.

? Frage: Warum ist der Button [Next] im Refactoring-Wizard ausgegraut?

! Antwort: Die Auswahl bzw. Angabe des Anwenders entspricht nicht der definierten Bedingung aus der Konfigurationsdatei zur Fortführung des Refactorings. In der Konfigurationsdatei sollte zu jedem Refactoring-Wizard-Dialog eine Beschreibung des Fehlers angegeben werden, der die Fortführung des Refactorings verhindert. Diese Beschreibung sollte in der `DecisionValidationDescription` der Konfigurationsdatei angegeben sein und wird im Wizard gegebenenfalls eingeblendet.

5 Tutorial

„Super, mach' doch mal. 15 Minuten haben wir!“

Tobias, über das Tutorial „Learning Eclipse in 15 minutes“

5.1 Einleitung

In diesem Kapitel wird anhand eines durchgehenden Beispiels die Nutzung und Funktionsweise der Komponenten Editor, Modellqualitätsplan, Diagnose sowie Refactoring beschrieben. Für diese wird der Anwender jeweils Schritt für Schritt durch die notwendigen Aktionen im Erstellungs- bzw. Verwendungsprozess geführt. Das Tutorial stützt sich dabei auf Beispiel-Abbildungen der Software, die einzelne bzw. zusammengehörige Schritte illustrieren und den Anwender beim Einstieg in die Verwendung der Software unterstützen sollen.

5.2 Durchgehendes Beispiel

In den folgenden Kapiteln wird ausgehend von der Erstellung eines ersten UML/Z-Modells über die Modellierung von Qualität mit Hilfe von Modellqualitätsplänen bis hin zur Diagnose und anschließender Durchführung eines Refactorings ein typischer Verwendungsablauf von RMC dargestellt.

Die Reihenfolge der im Tutorial beschriebenen Arbeitsschritte ist hierbei exemplarisch für eine erste Verwendung von RMC. Im weiteren Verlauf der Arbeit mit RMC können Schritte gegebenenfalls teilweise oder im Ganzen übersprungen werden. Mehr zum Thema Qualitätskreislauf findet der interessierte Leser in Kapitel 2.2.

5.2.1 Editor

Zunächst wird ein UML/Z-Modell benötigt. Die zur Erstellung des Modells notwendigen Schritte lassen sich wie folgt aufzählen:

- Neues Projekt anlegen
- Benötigte Datentypen erstellen

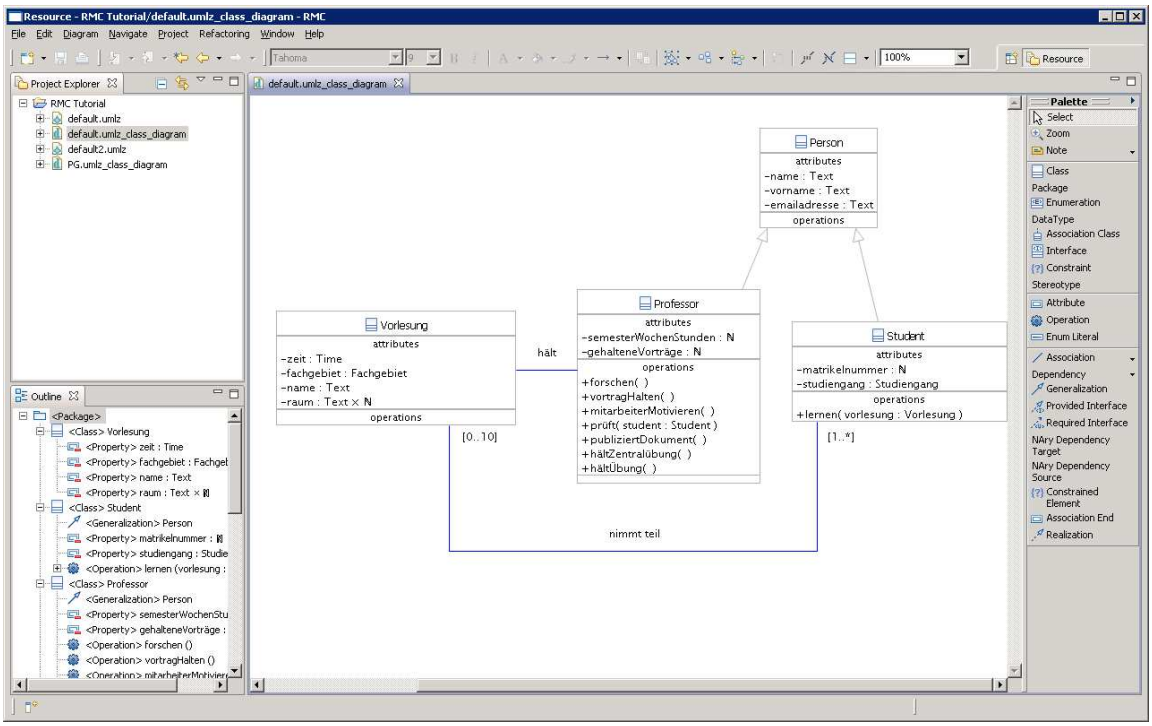


Abbildung 5.1: Tutorial: Fertiges UML/Z-Modell

- Klassen mit ihren Attributen und Operationen anlegen
- Beziehungen der Klassen modellieren

Resultat wird schließlich das in Abbildung 5.1 dargestellte Modell sein.

5.2.1.1 Projekt anlegen

Als Erstes wird ein **Projekt** über ⟨`File`⟩→⟨`New`⟩→⟨`Project`⟩ angelegt. Der erscheinende Dialog (Abbildung 5.2) wird mit [Next] bestätigt.

Im nächsten Schritt wird ein Name für das neue Projekt festgelegt. Abbildung 5.3 zeigt die Eingabemaske zu diesem Vorgang. Der Anwender gibt **RMC Tutorial** ein und bestätigt mit [Finish].

Nun können dem Projekt Diagramme hinzugefügt werden. Über die Menüpunkte ⟨`File`⟩→⟨`New`⟩→⟨`Other`⟩→⟨`UMLZ Classdiagramm`⟩ wird ein neues Klassendiagramm angelegt. Abbildung 5.4 zeigt diesen Schritt.

Bestätigt der Anwender mit [Next], so wird die Eingabemaske aus Abbildung 5.5 angezeigt.

Hier wird ein Name für das neue Klassendiagramm definiert, wobei die Vorauswahl beibehalten werden kann. Das Anlegen wird schließlich mit [Finish] abgeschlossen.

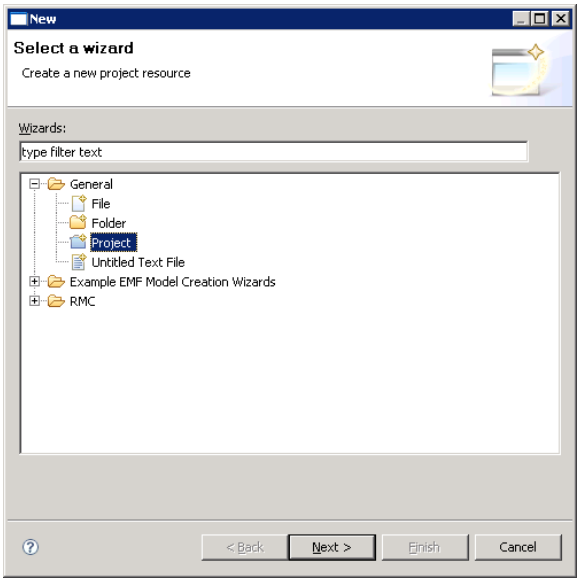


Abbildung 5.2: Tutorial: Projekt anlegen

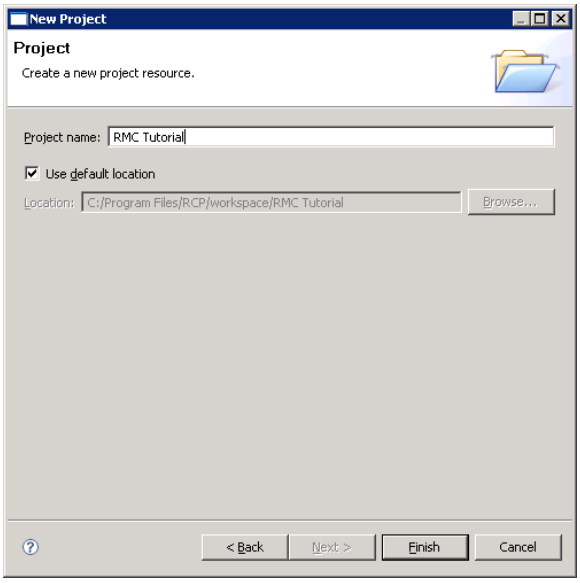


Abbildung 5.3: Tutorial: Projektname

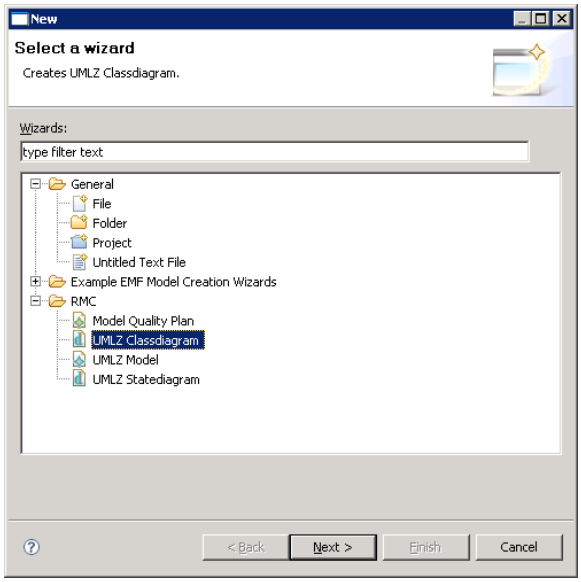


Abbildung 5.4: Tutorial: Diagrammtyp

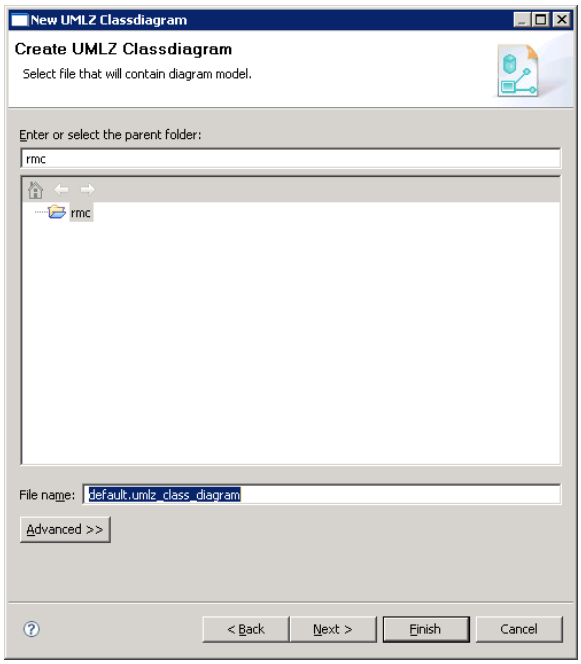


Abbildung 5.5: Tutorial: Dateiname für das Diagramm

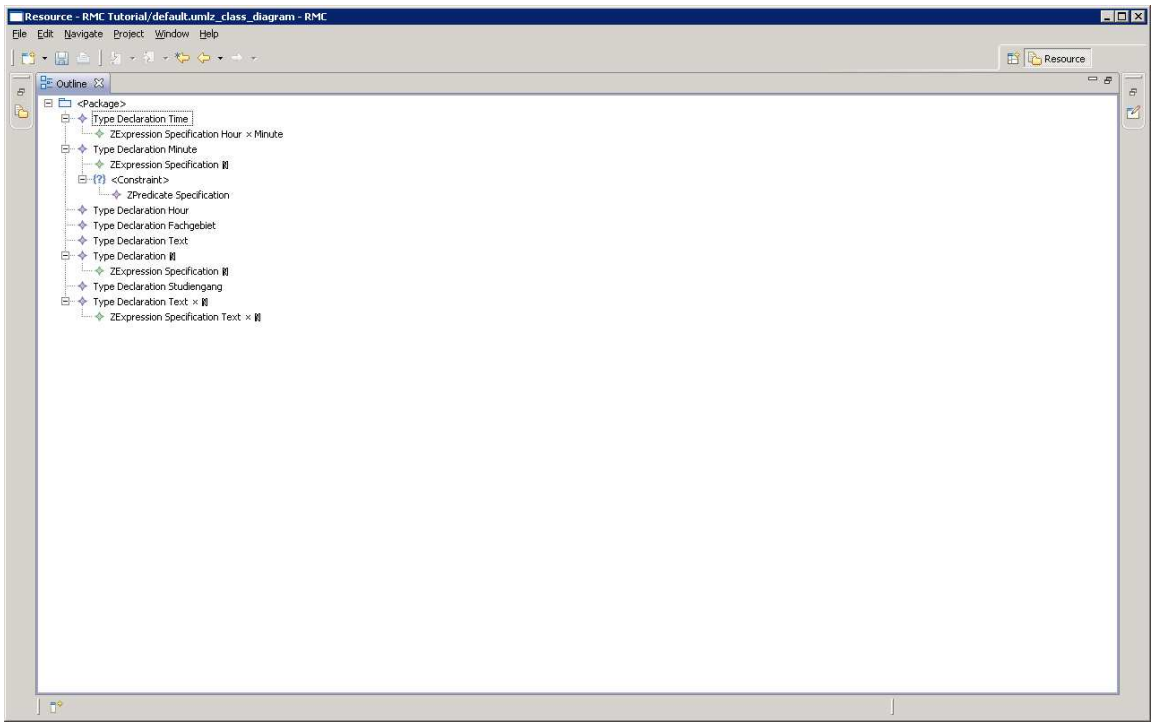


Abbildung 5.6: Tutorial: Elemente in der Outline

5.2.1.2 Datentypen erstellen

Bei der Erstellung von Datentypen in Diagrammen wird im UML/Z-Editor zwischen zwei **Typen** von Elementen unterschieden:

- Elemente, die über die Outline angelegt werden
- Elemente, die auf der Zeichenfläche angelegt werden

Datentypen werden über die Outline angelegt. Am Ende dieses Kapitels wird die Outline wie in Abbildung 5.6 dargestellt aussehen.

Das Modell dieses Tutorials soll acht Elemente vom Typ **Type Declaration** enthalten. Diese werden nun Schritt für Schritt angelegt:

Type Declaration Time: In der Outline wird mit einem Rechtsklick auf **Package** das Kontextmenü geöffnet. Mit ⟨`RMC`⟩→⟨`Packaged Element`⟩→⟨`Type Declaration`⟩ wird ein neuer Datentyp angelegt. Für diesen wird der Name **Time** eingegeben. Die Eigenschaften dieses Elementes entsprechen nun der Abbildung 5.7. Weiter wird ein neues Kindelement angelegt. Auf dem neuen Element wird wieder im Kontextmenü ⟨`RMC`⟩→⟨`ZExpression Specification`⟩ gewählt. Abbildung 5.8 zeigt die Eigenschaften des neuen Elementes. Mit einem Klick auf den [...] Button wird der **Z-Editor** geöffnet und der folgende Ausdruck eingegeben: `Hour \cross Minute`. Abbildung 5.9 zeigt den ZExpression Editor. Mit [OK] wird der Editor schließlich verlassen.

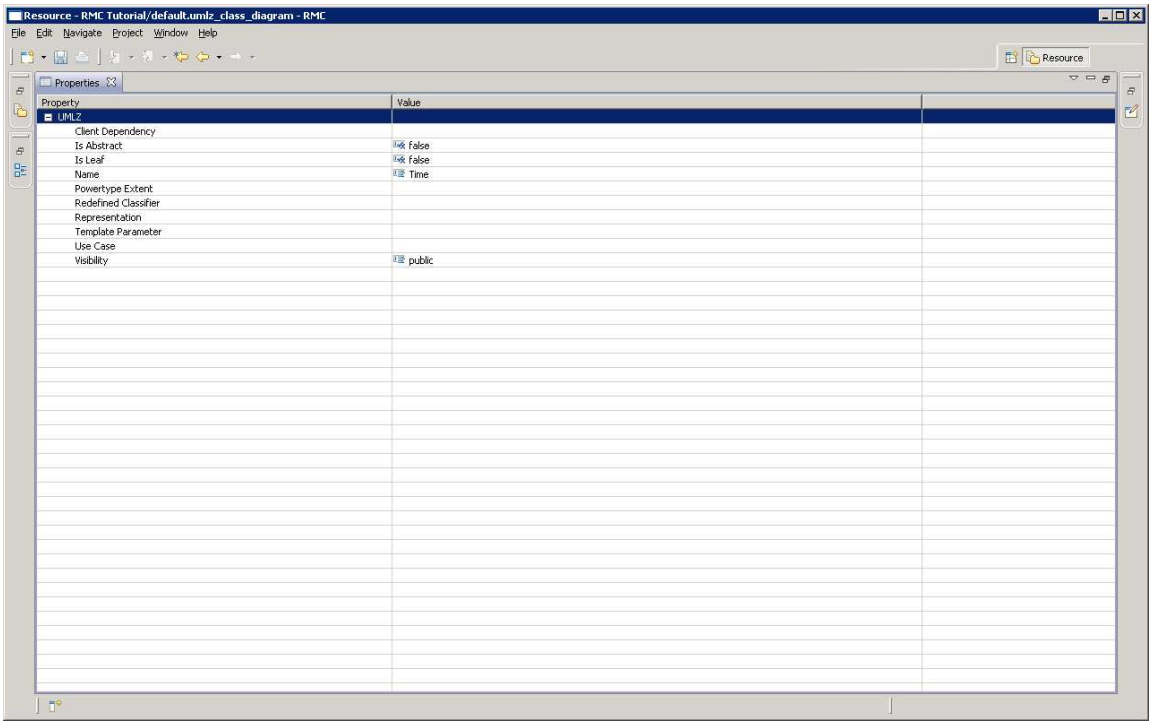


Abbildung 5.7: Tutorial: Eigenschaften des Datentyps Time

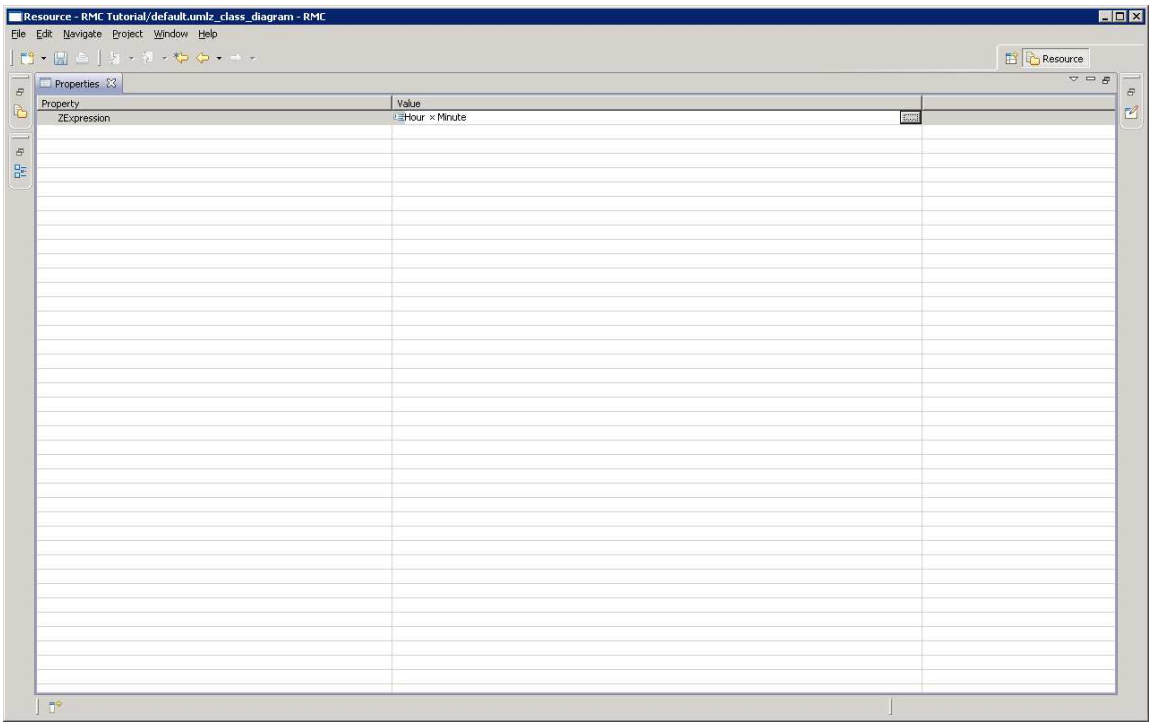


Abbildung 5.8: Tutorial: Eigenschaften der ZExpression Specification für Time

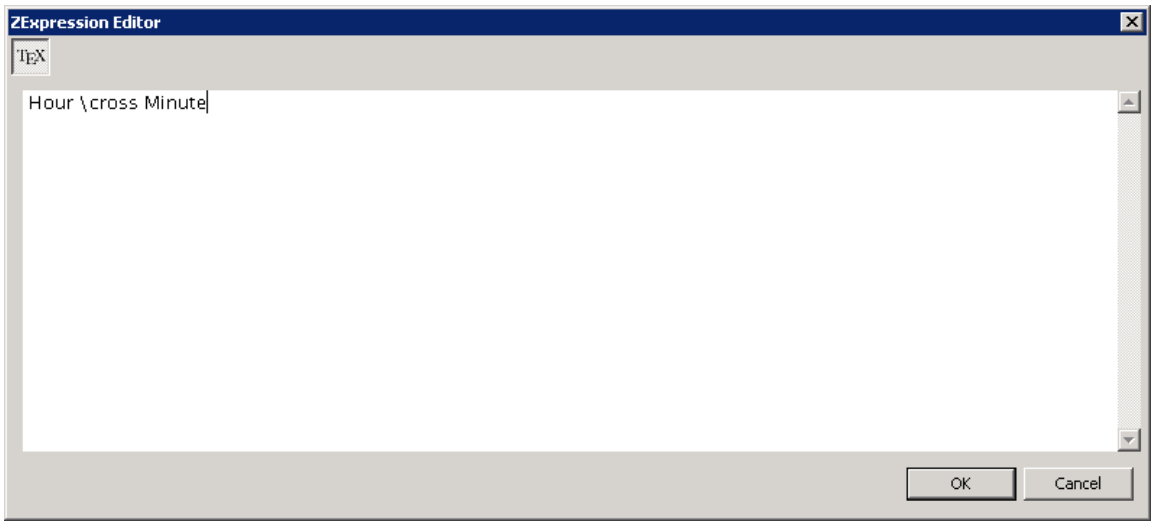


Abbildung 5.9: Tutorial: Z-Editor - ZExpression für Time

Type Declaration Minute: Als nächstes Element wird eine **Type Declaration** mit dem Namen **Minute** angelegt. Abbildung 5.10 zeigt die Eigenschaften dieses Elementes. Wieder wird für das neue Element mit Hilfe des Kontextmenüs ein Kind vom Typ **ZExpression Specification** erstellt. Der Z-Editor wird gestartet und `\nat` eingegeben. Abbildung 5.11 zeigt diese Eingabe im Editor. Analog wird nun noch ein Element vom Typ Constraint auf der **Type Declaration** angelegt, Abbildung 5.12 zeigt dessen Eigenschaften. Dieser Constraint erhält eine **ZPredicate Specification** mit folgendem Z-Ausdruck: Minute $= \langle\langle 0\,..\,59 \rangle\rangle$ Dies ist in Abbildung 5.13 dargestellt.

Type Declaration Hour: Als nächster Datentyp wird **Hour** angelegt. Dieser wird, wie bereits zuvor, mit dem ⟨RMC⟩-Menü angelegt und erhält **Hour** als Name. Abbildung 5.14 zeigt die Eigenschaften des Elementes.

Type Declaration Fachgebiet: Als weiterer Datentyp wird **Fachgebiet** erzeugt. Analog zu den vorherigen Schritten wird dieser über das ⟨RMC⟩-Menü angelegt und erhält **Fachgebiet** als Name. Abbildung 5.15 zeigt die zugehörigen Eigenschaften.

Type Declaration Text: Die gleiche Vorgehensweise wird für den Datentyp **Text** angewendet. Die Erstellung geschieht wieder über das ⟨RMC⟩-Menü, als Name wird **Text** angegeben. Abbildung 5.16 zeigt die Eigenschaften.

Type Declaration $\mathbb{N}$: Als nächster Datentyp wird $\mathbb{N}$ mit Hilfe des ⟨RMC⟩-Menü erstellt, wobei als Name $\mathbb{N}$, wie in Abbildung 5.17 gezeigt, eingeben wird.

Type Declaration Studiengang: Die Vorgehensweise zur Erstellung des Datentyps **Studiengang** geschieht analog unter Angabe des Namens **Studiengang**. 5.18 zeigt das Ergebnis.

Type Declaration $Text \times \mathbb{N}$: Als letzter Datentyp wird $Text \times \mathbb{N}$ erstellt. Die Erstellung gleicht den vorherigen Schritten unter Angabe des Namens $Text \times \mathbb{N}$ mit dem Ergebnis in Abbildung 5.19.

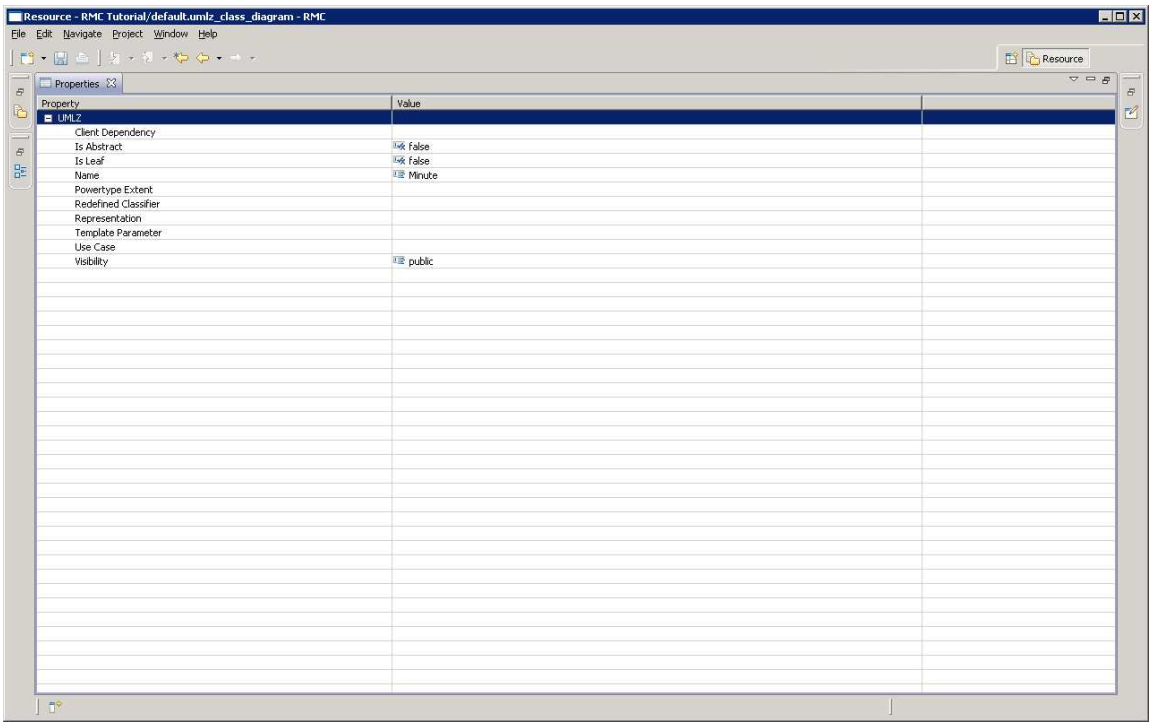


Abbildung 5.10: Tutorial: Eigenschaften des Datentyps Minute

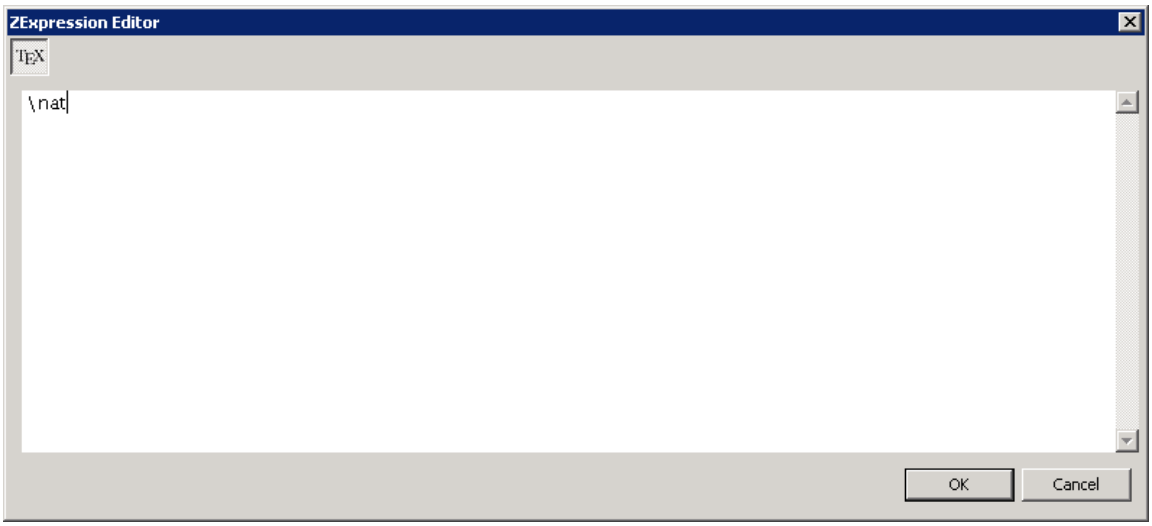


Abbildung 5.11: Tutorial: Z-Editor - ZExpression für Minute

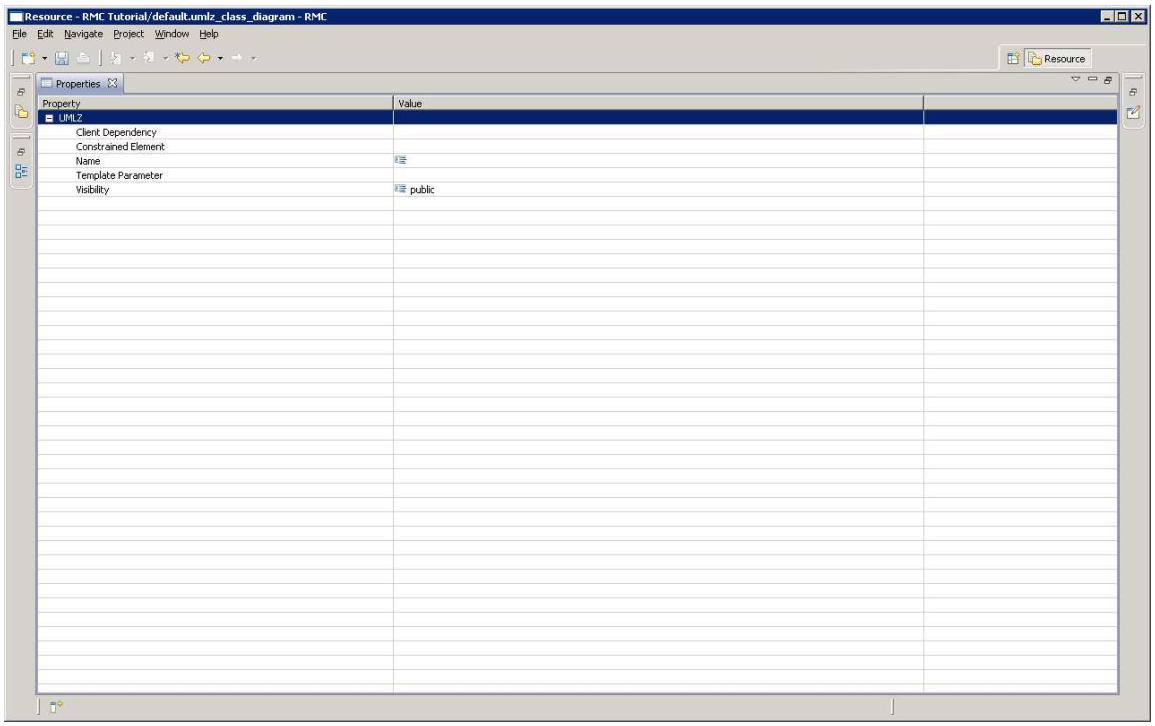


Abbildung 5.12: Tutorial: Constraint für Minute

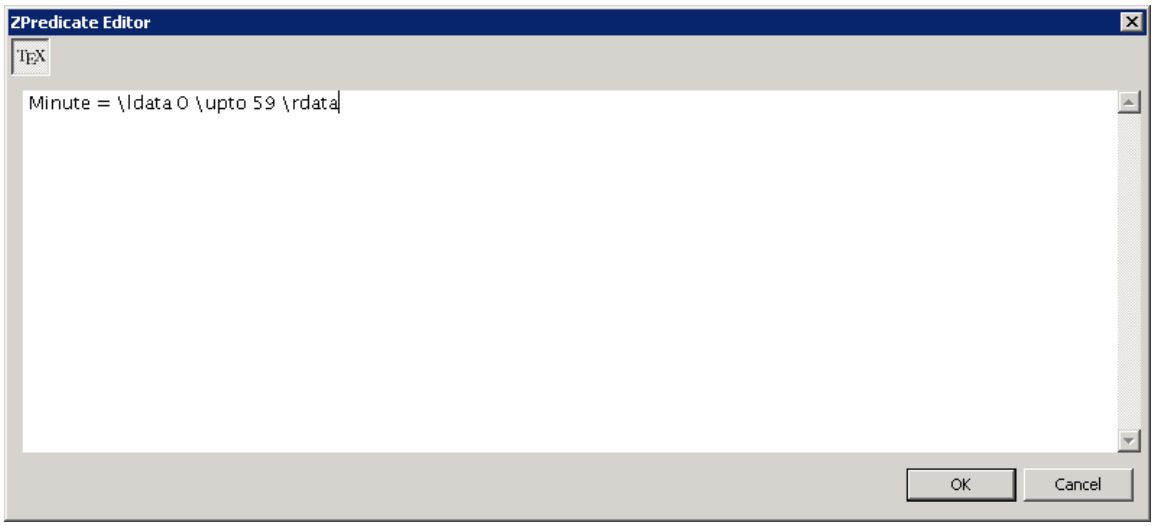


Abbildung 5.13: Tutorial: Z-Editor - ZPredicate für Minute

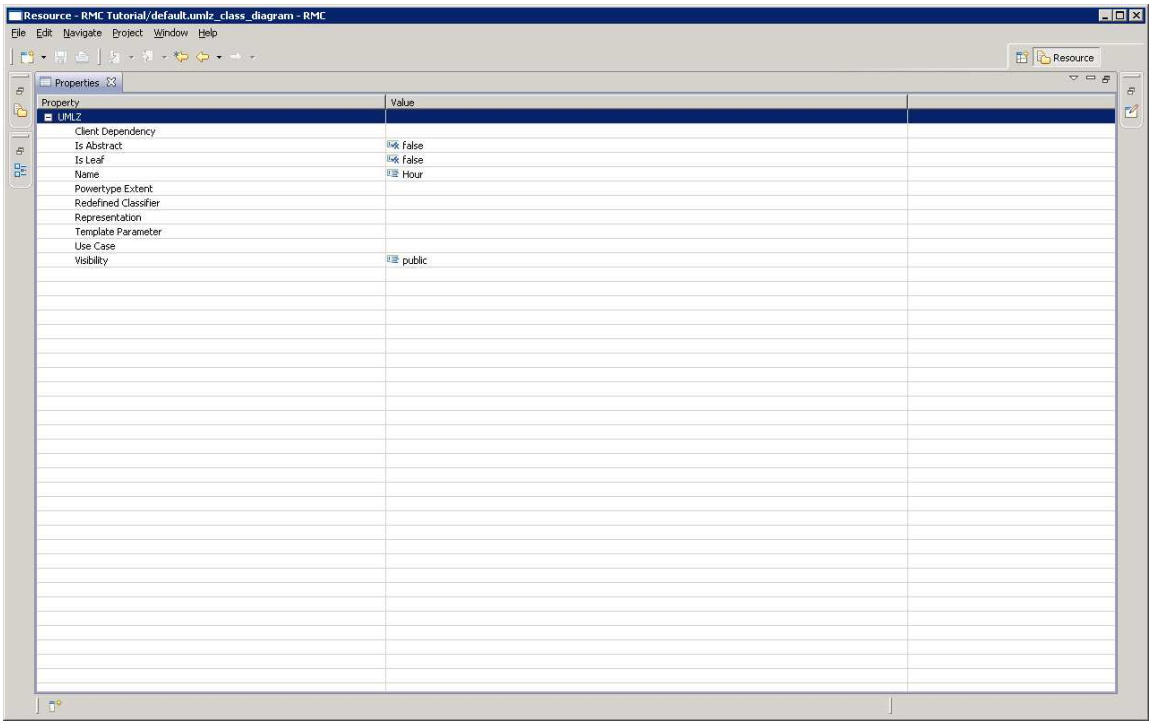


Abbildung 5.14: Tutorial: Eigenschaften des Datentyps Hour

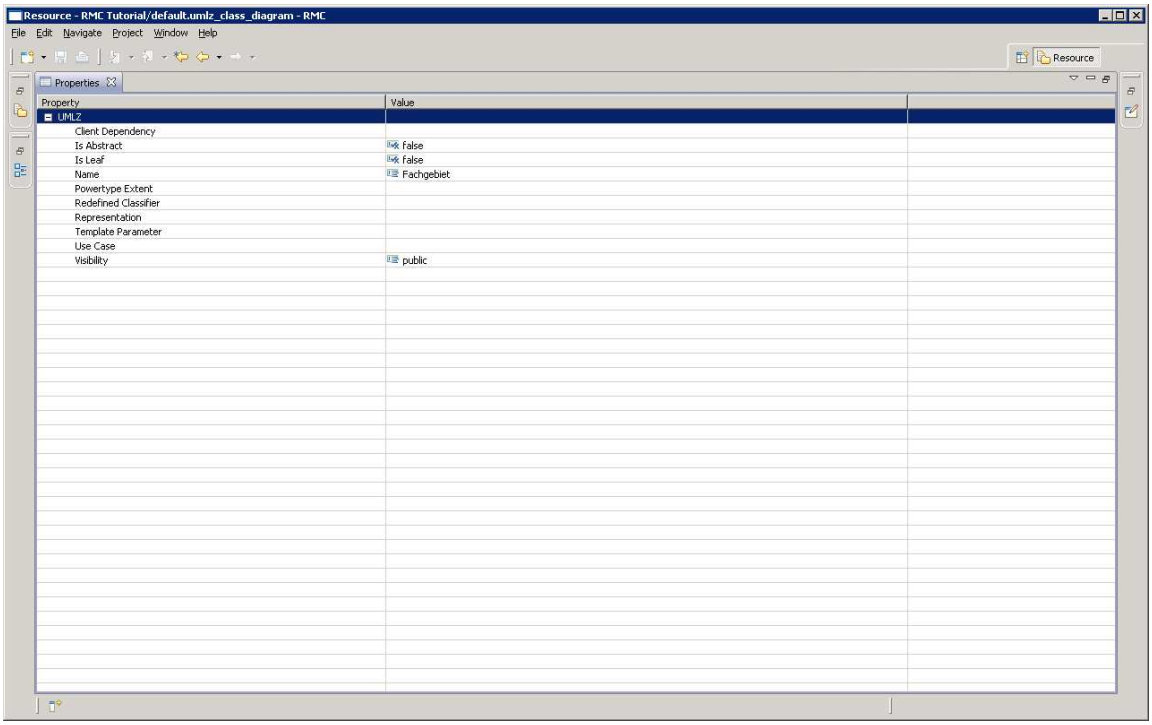


Abbildung 5.15: Tutorial: Eigenschaften des Datentyps Fachgebiet

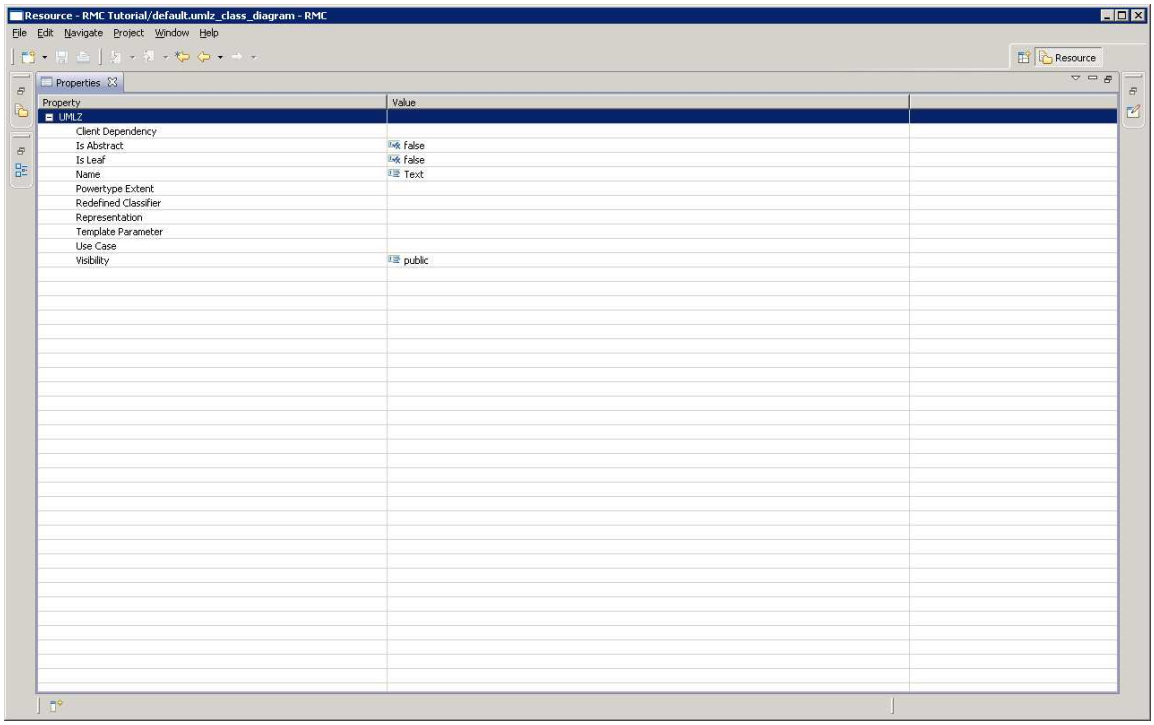


Abbildung 5.16: Tutorial: Eigenschaften des Datentyps Text

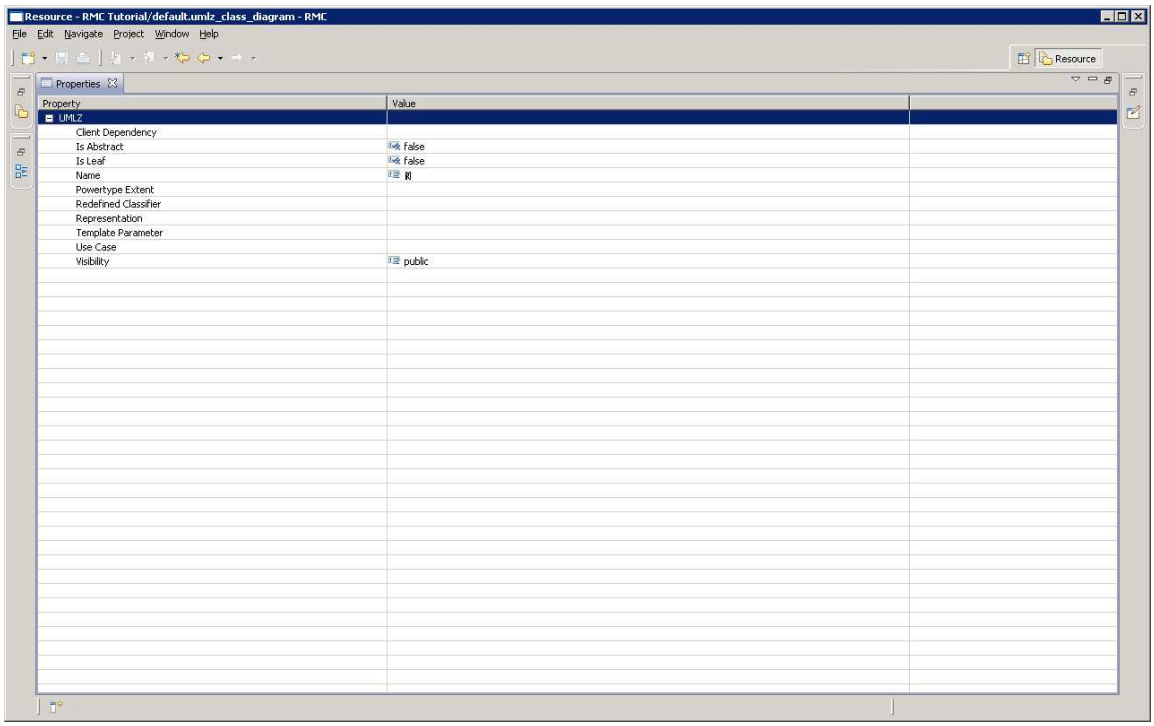


Abbildung 5.17: Tutorial: Eigenschaften des Datentyps ℕ

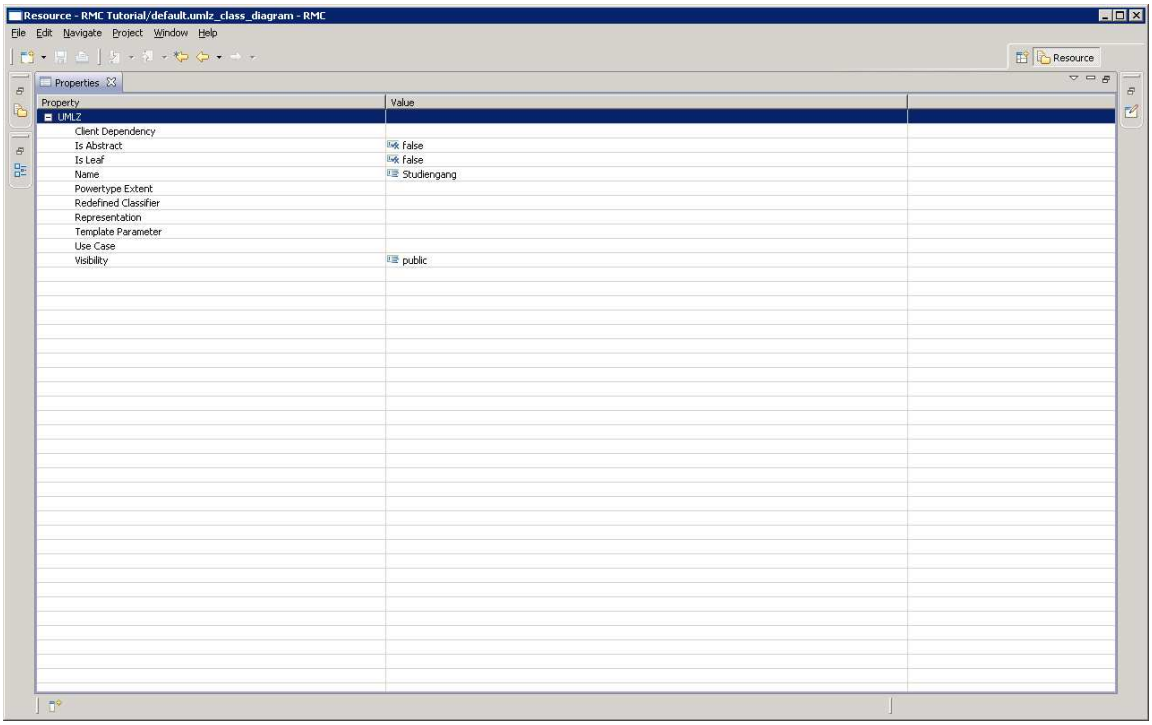


Abbildung 5.18: Tutorial: Eigenschaften des Datentyps Studiengang

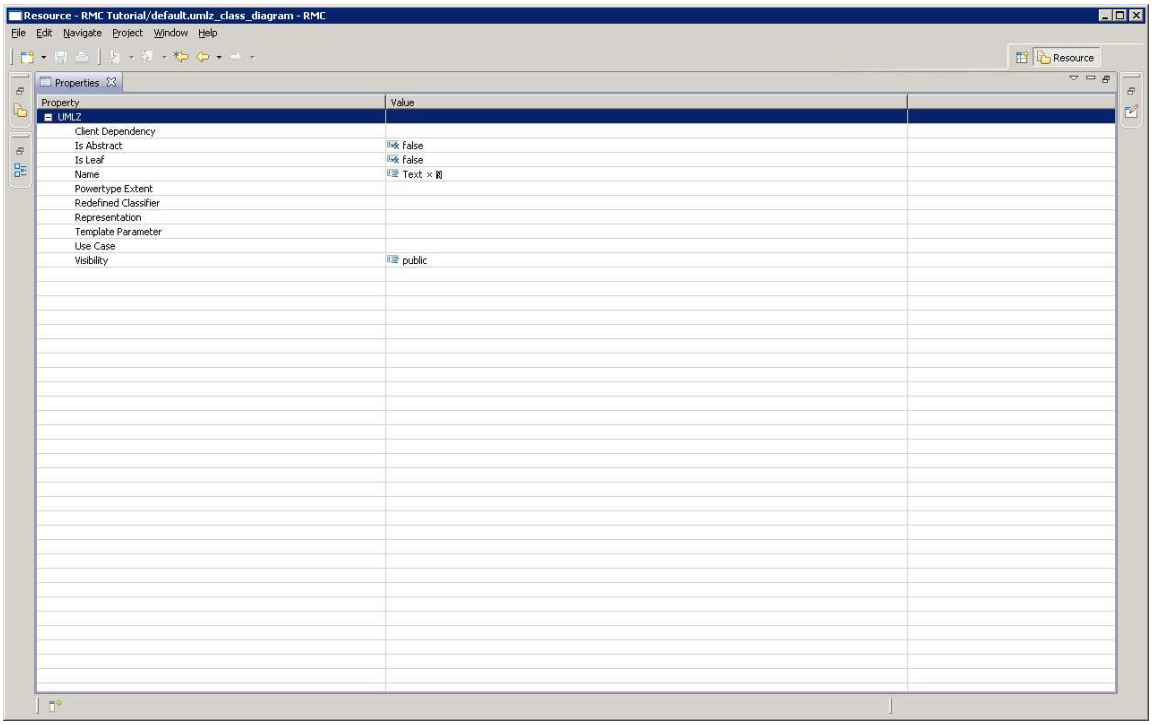


Abbildung 5.19: Tutorial: Eigenschaften des Datentyps $\mathit{Text} \times \mathbb{N}$

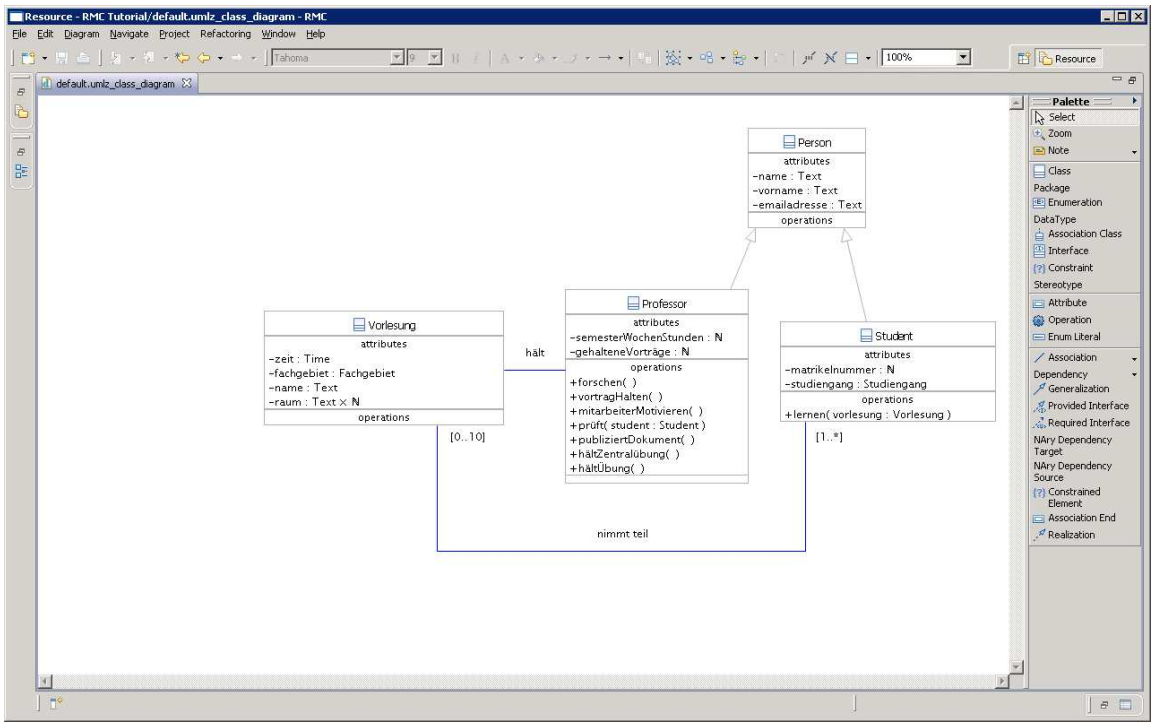


Abbildung 5.20: Tutorial: Die Zeichenfläche des Editors

5.2.1.3 Klassen anlegen

In diesem Kapitel wird die Erzeugung von Elementen, die über die Zeichenfläche angelegt werden, demonstriert. Abbildung 5.20 zeigt die resultierende Zeichenfläche des Editors mit allen Elementen, die im Folgenden erstellt werden.

Es werden vier Klassen, ihre Attribute und Operationen angelegt:

Die Klasse Person: Um die erste Klasse anzulegen, wird in der Palette rechts neben der Zeichenfläche das Werkzeug **Class** ausgewählt. Mit einem Klick in die Zeichenfläche wird eine neue Klasse angelegt. In den Eigenschaften wird für sie der **Name Person** eingegeben. Abbildung 5.21 zeigt die Eigenschaften der Klasse.

Nun werden dieser Klasse noch drei Attribute hinzugefügt. Hierzu wird aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Person geklickt. Hierdurch wird der Klasse das Attribut hinzugefügt. Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **name** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox für das Feld Typ **Text** gewählt wird. Abbildung 5.22 zeigt das Resultat.

Um das zweite Attribut zu erstellen, wird wieder aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Person geklickt.

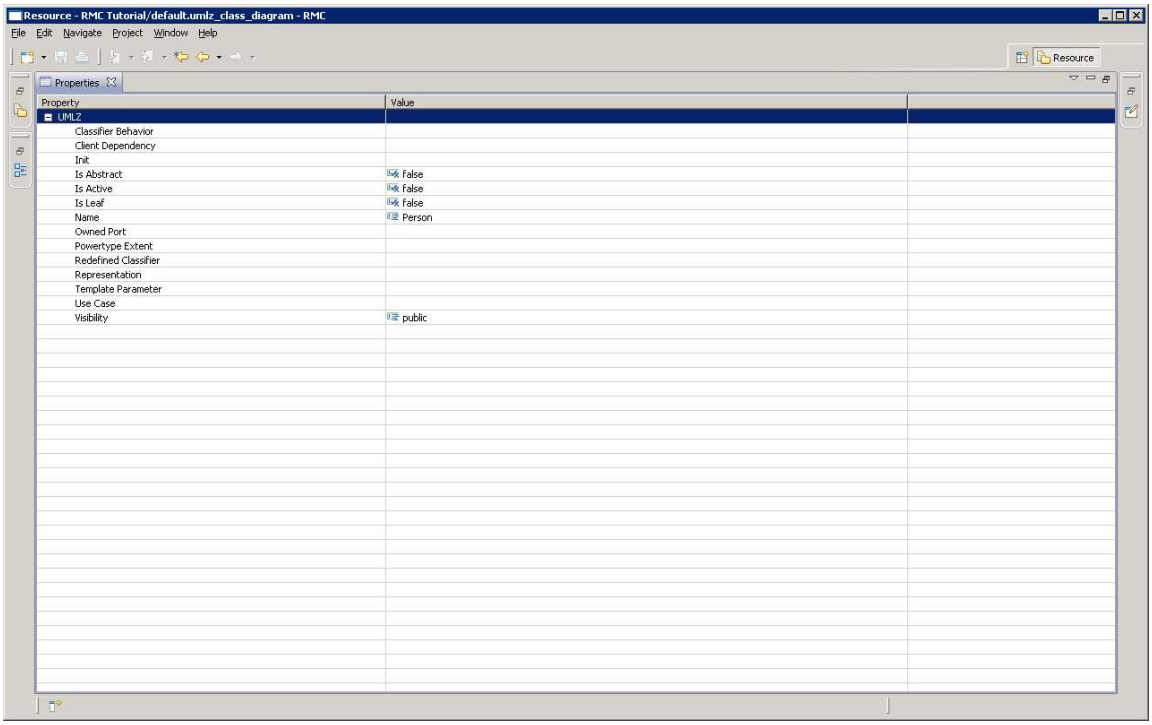


Abbildung 5.21: Tutorial: Eigenschaften der Klasse Person

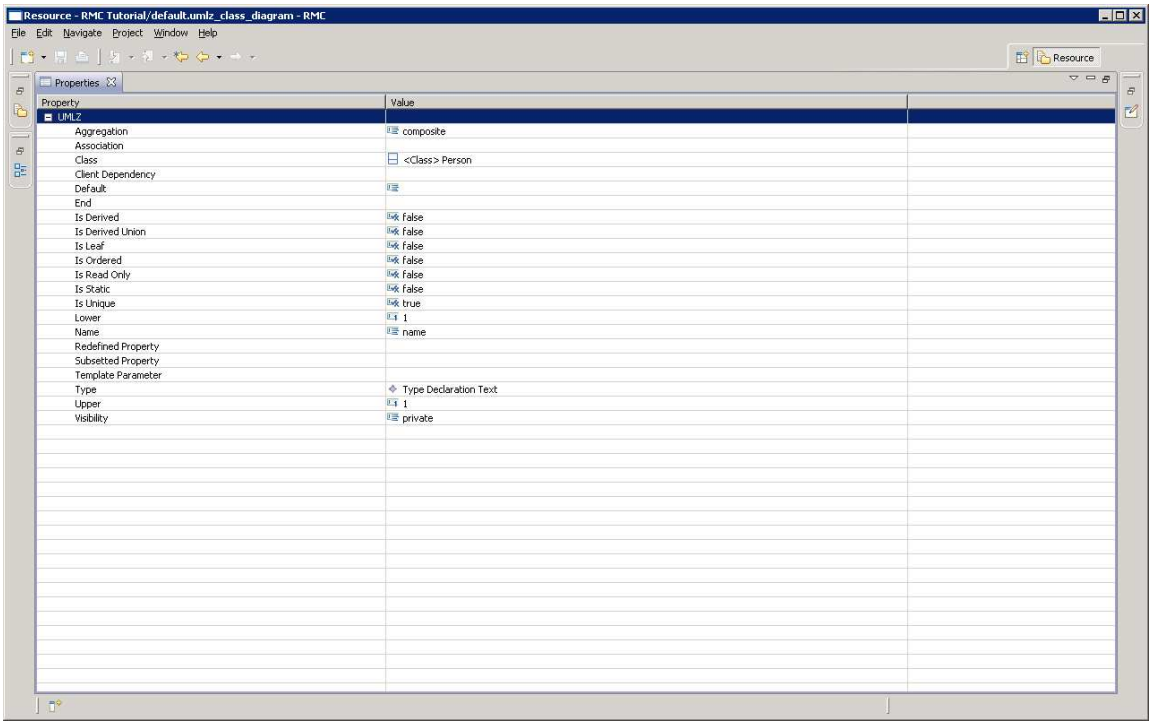


Abbildung 5.22: Tutorial: Eigenschaften des Attributes name

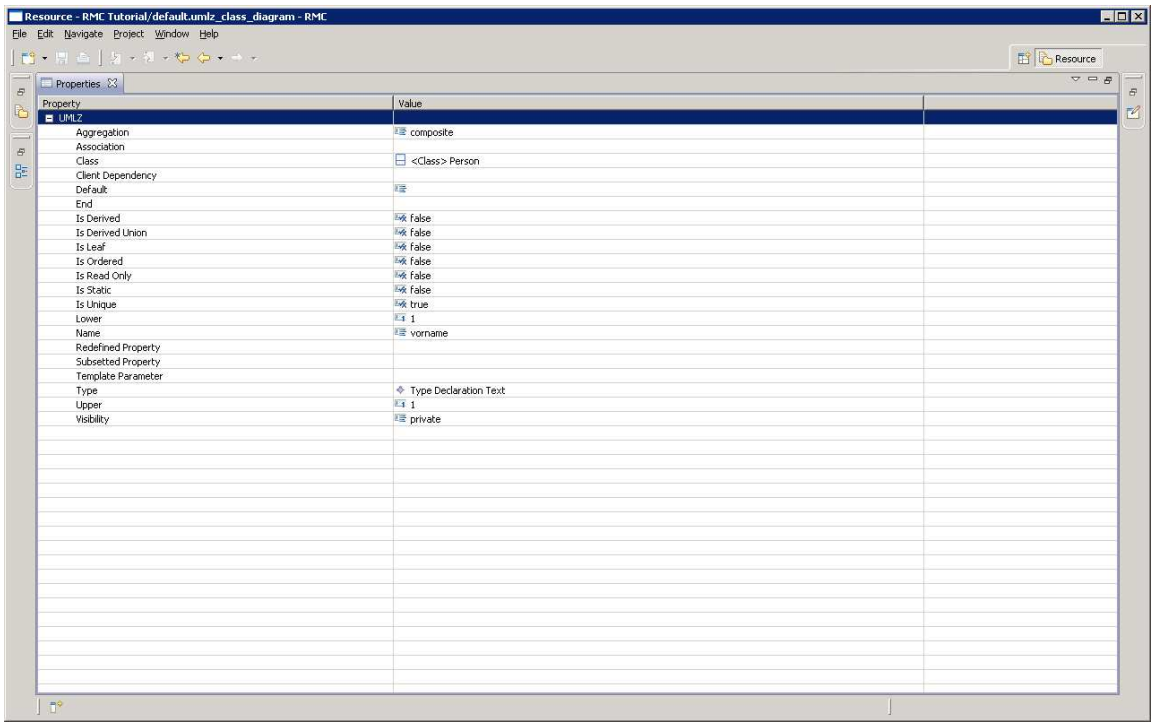


Abbildung 5.23: Tutorial: Eigenschaften des Attributes vorname

Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **vorname** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox für das Feld Typ **Text** gewählt wird. Abbildung 5.23 zeigt das Resultat.

Das dritte Attribut wird erstellt. Nach der Auswahl des Werkzeuges **Attribute** und Klick auf die Klasse Person wird der Name des Attributes, hier **emailadresse**, in der Properties View im Feld name eingetragen. Ebenfalls über die Property-View wird wie zuvor der Typ des Attributes auf **Text** gesetzt. Abbildung 5.24 zeigt das Resultat.

Die Klasse Student: Analog zu der Klasse Person wird die Klasse Student angelegt, indem in der Palette rechts neben der Zeichenfläche das Werkzeug **Class** ausgewählt wird. Mit einem Klick in die Zeichenfläche erscheint dort die neue Klasse. In den Eigenschaften dieser wird der **Name Student** eingegeben. Abbildung 5.25 zeigt die Eigenschaften der Klasse.

Nun werden dieser Klasse noch zwei Attribute hinzugefügt. Hierzu wird aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Student geklickt. Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **matrikelnummer** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox

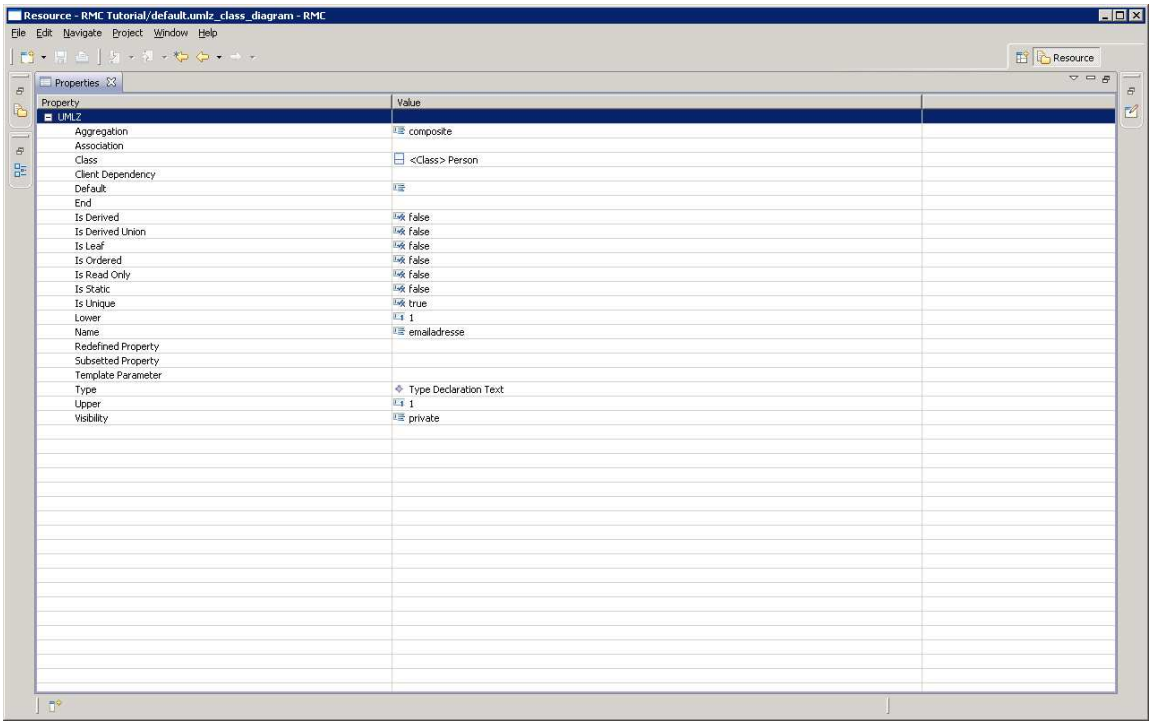


Abbildung 5.24: Tutorial: Eigenschaften des Attributes email

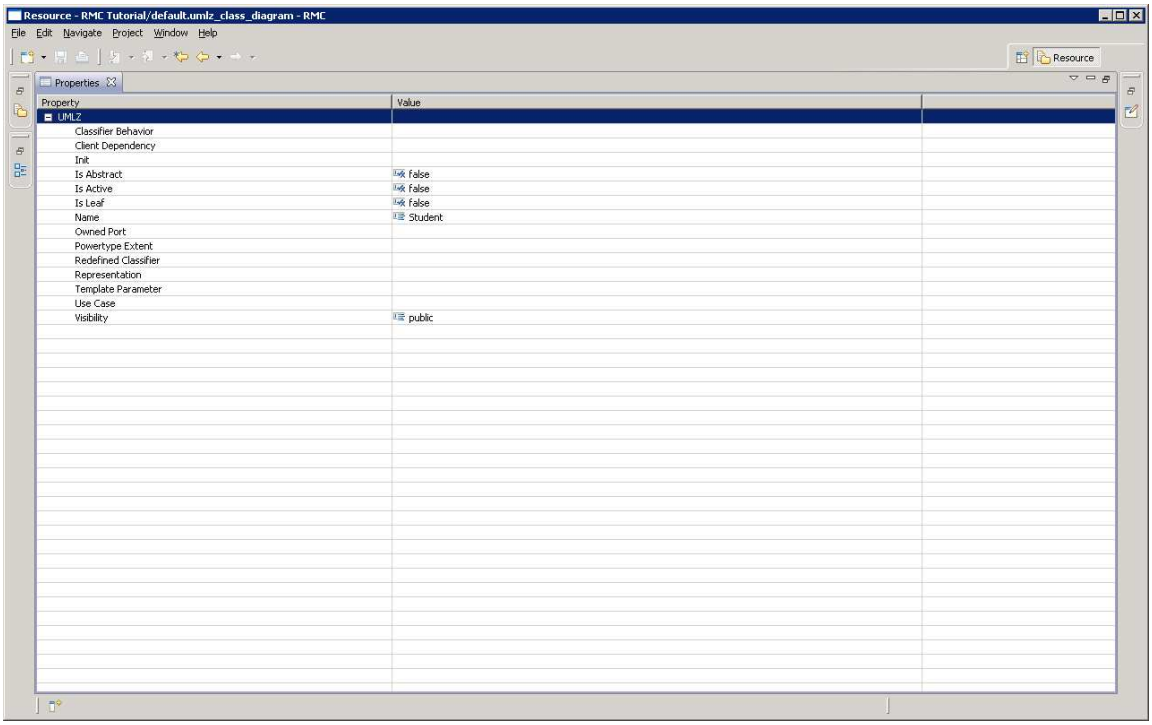


Abbildung 5.25: Tutorial: Eigenschaften der Klasse Student

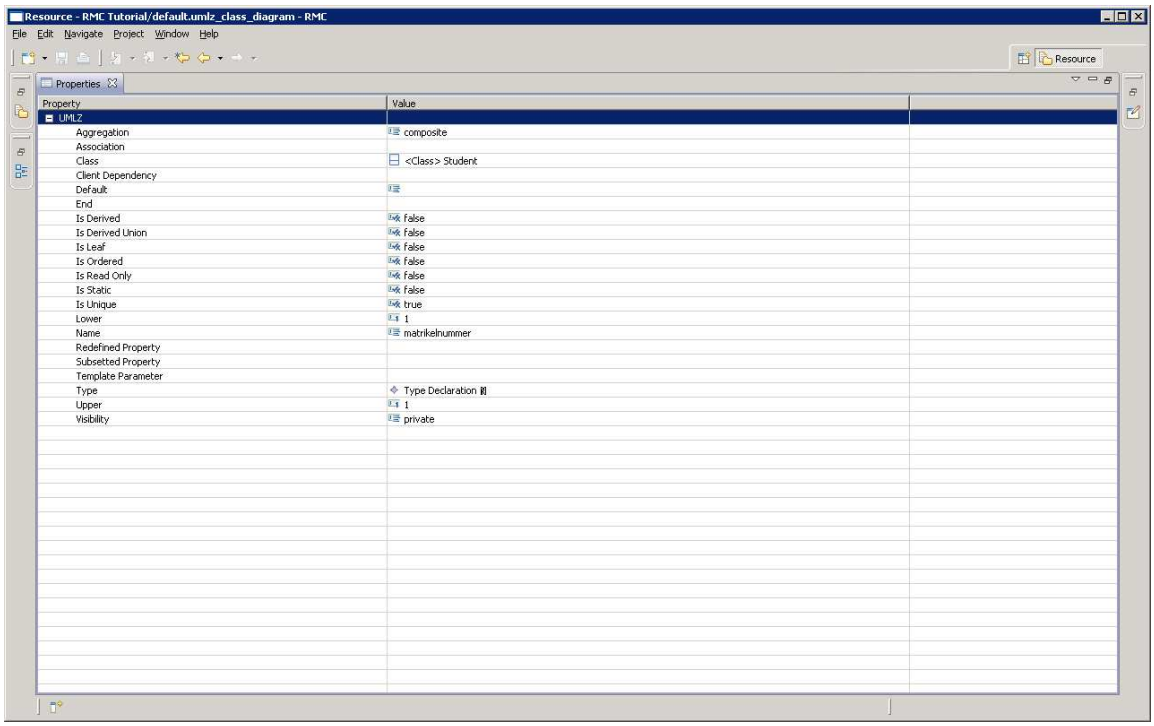


Abbildung 5.26: Tutorial: Eigenschaften des Attributes Matrikelnummer

für das Feld Typ $\mathbb{N}$ gewählt wird. Abbildung 5.26 zeigt das Resultat.

Um das zweite Attribut zu erstellen, wird wieder aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Student geklickt. Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **studiengang** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox für das Feld Typ **Studiengang** gewählt wird. Abbildung 5.27 zeigt das Resultat.

Nun muss eine Operation zur Klasse Student hinzugefügt werden. Hierzu wird aus der Palette das Werkzeug **Operation** gewählt und durch Klick auf die Klasse Student dieser eine Operation hinzugefügt. Um den Namen der Operation anzugeben, wird die Operation in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **lernen** eingetragen. Nun muss noch der Parameter **vorlesung** angegeben werden. Dies geschieht mit Hilfe der Outline. Zunächst wird die Operation in der Zeichenfläche durch Klick ausgewählt. In der Outline erscheint nun die Operation. Hier wird durch Rechtsklick auf die Operation das Kontextmenü geöffnet und über ⟨`Create child`⟩→⟨`Parameter`⟩ der Parameter für die Operation erstellt. Durch Klick auf den neu erstellten Parameter in der Outline wird die zugehörige Properties-View geöffnet und dort für das Feld name **vorlesung** eingetragen. Weiter muss der Typ des Parameters, hier Vorlesung, angegeben werden. Durch Auswahl des Typs **Vorlesung** in der Property-View des gerade erzeugen Parameters wird dies umgesetzt. Abbildung 5.28 zeigt das Resultat.

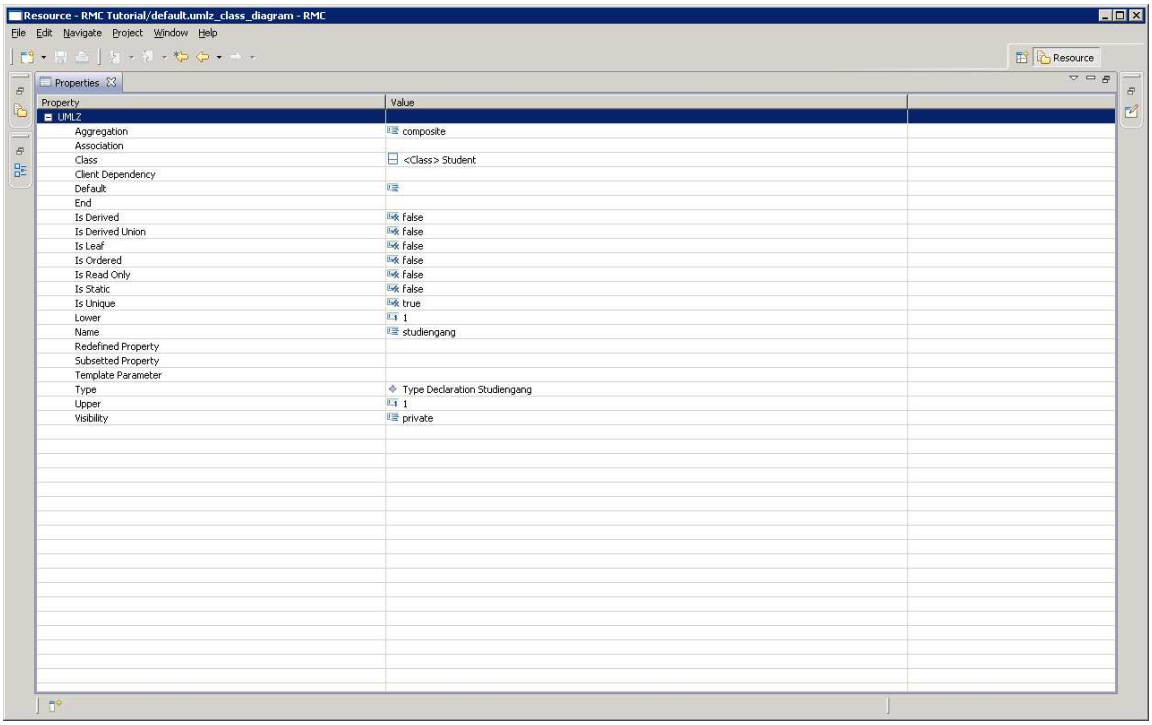


Abbildung 5.27: Tutorial: Eigenschaften des Attributes Studiengang

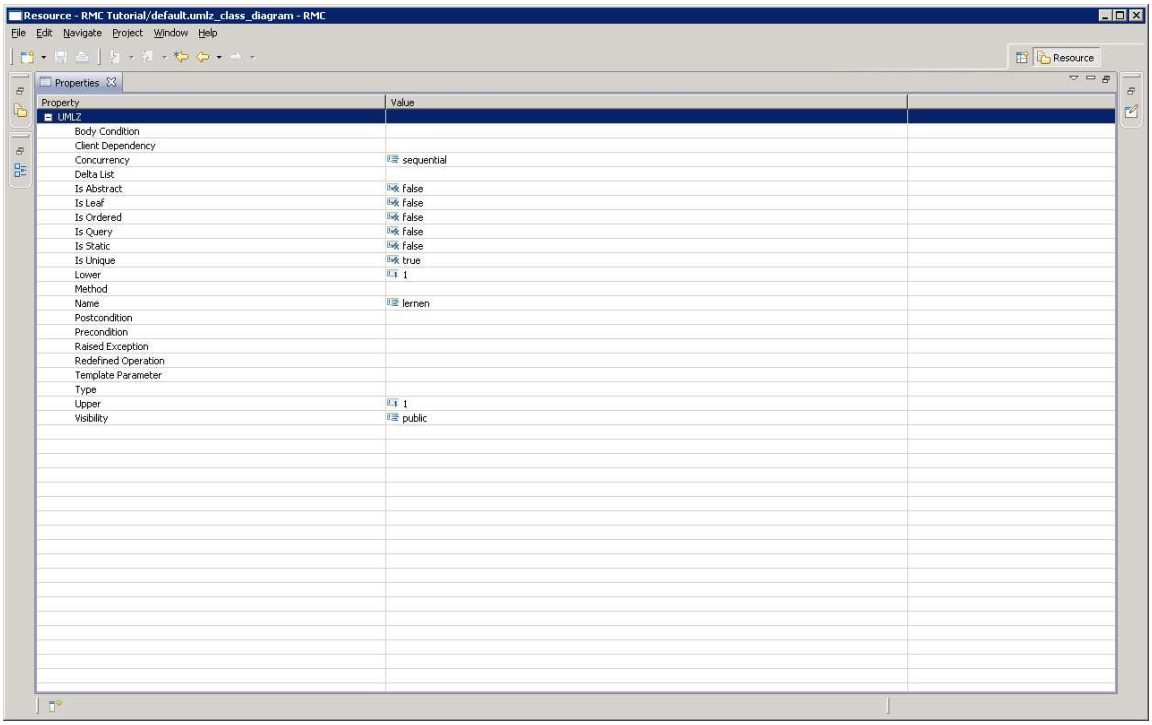


Abbildung 5.28: Tutorial: Eigenschaften der Methode lernen

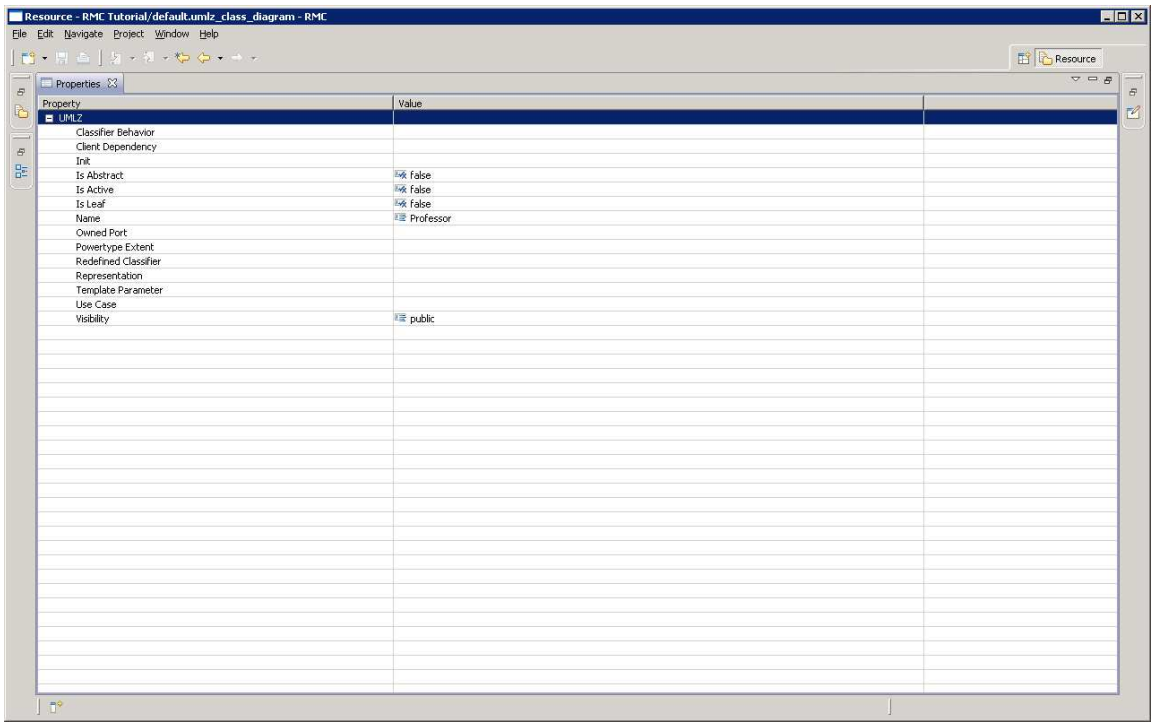


Abbildung 5.29: Tutorial: Eigenschaften der Klasse Professor

Die Klasse Professor: Nun wird die Klasse Professor angelegt. In der Palette wird das Werkzeug **Class** gewählt und durch einen Klick in die Zeichenfläche eingefügt. In den Eigenschaften der Klasse wird der **Name Professor** eingegeben. Abbildung 5.29 zeigt die zugehörigen Eigenschaften.

Nun werden dieser Klasse noch 2 Attribute hinzugefügt. Hierzu wird aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Professor geklickt. Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **semesterWochenStunden** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox für das Feld Typ $\mathbb{N}$ gewählt wird. Abbildung 5.30 zeigt das Resultat.

Um das zweite Attribut zu erstellen, wird wieder aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Professor geklickt. Um den Namen des Attributs anzugeben, wird das Attribut in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **gehalteneVorträge** eingetragen. Nun muss noch der Typ des Attributs angegeben werden. Dies geschieht ebenfalls in der Properties-View, indem aus der Auswahlbox für das Feld Typ $\mathbb{N}$ gewählt wird. Abbildung 5.31 zeigt das Resultat.

Als nächstes werden sieben Operationen für die Klasse Professor angelegt. Die Operation **forschen** wird erstellt: Hierzu wird aus der Palette das Werkzeug **Operation** gewählt und durch Klick auf die Klasse Professor dieser eine Operation hinzugefügt.

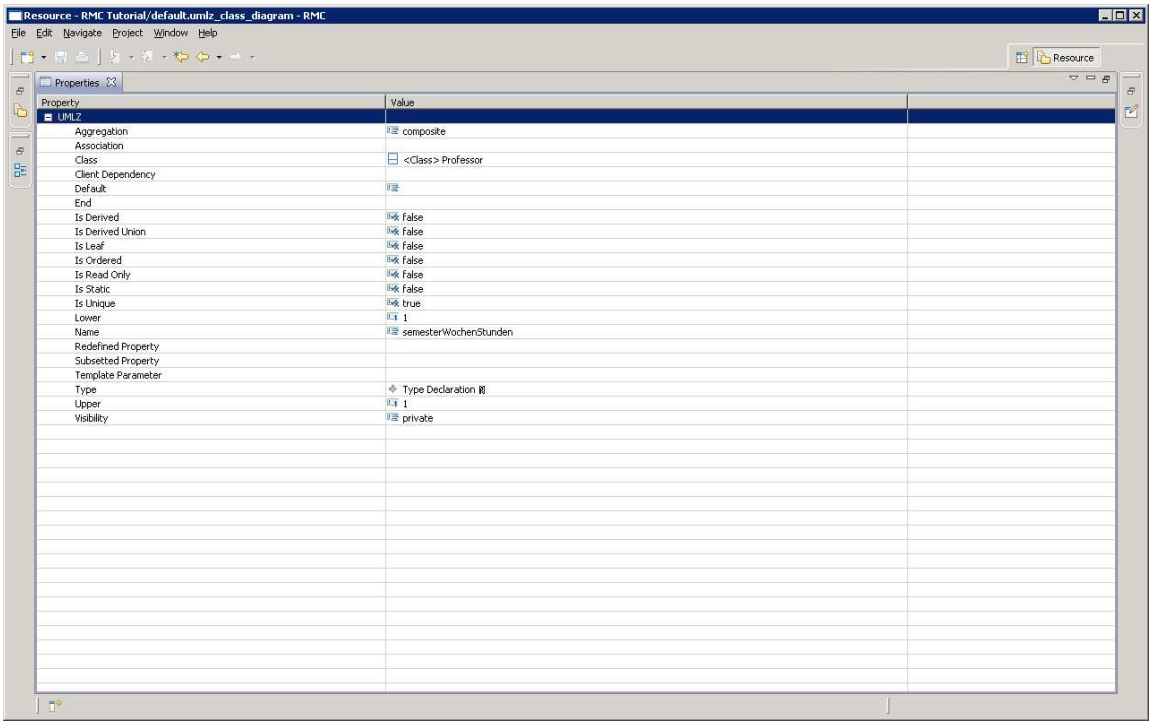


Abbildung 5.30: Tutorial: Eigenschaften des Attributes semesterWochenStunden

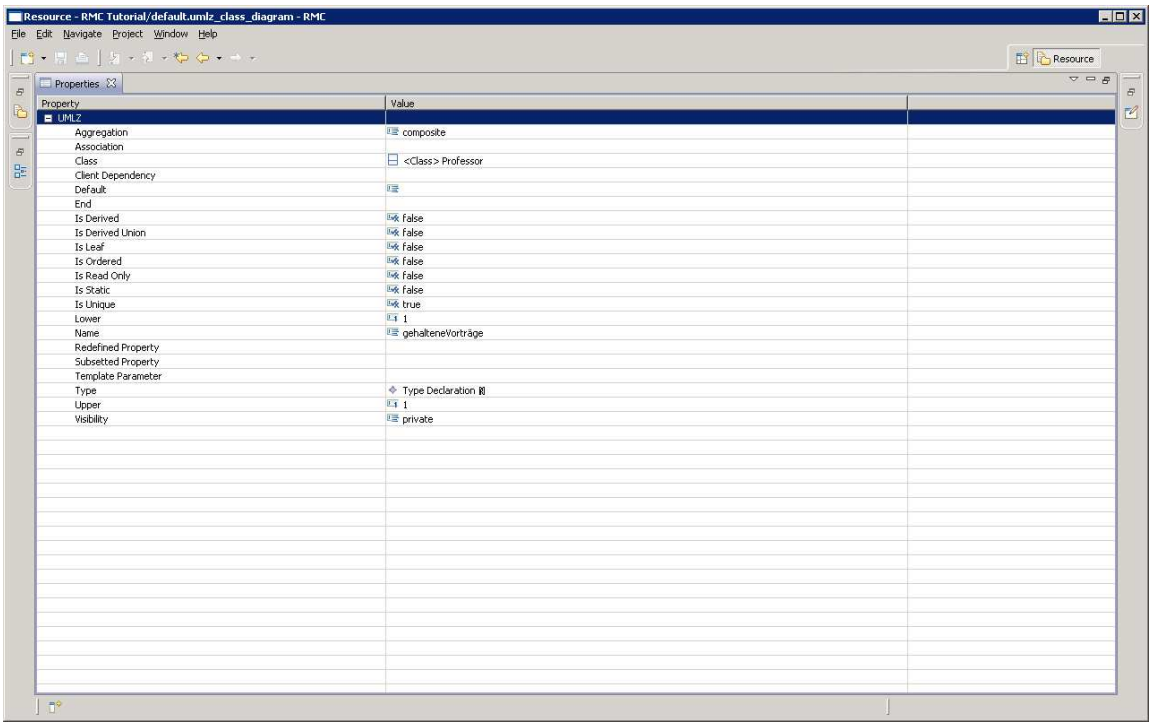


Abbildung 5.31: Tutorial: Eigenschaften des Attributes gehalteneVorträge

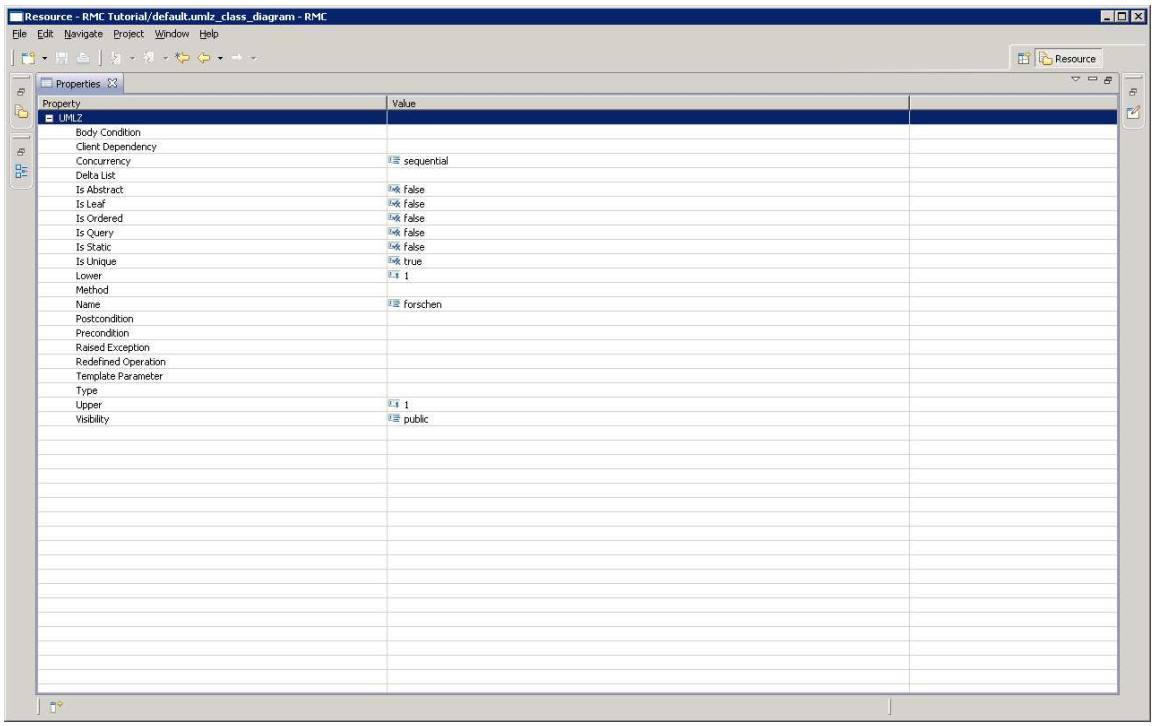


Abbildung 5.32: Tutorial: Eigenschaften der Methode forschen

Um den Namen der Operation anzugeben wird die Operation in der Zeichenfläche angeklickt und in der Properties-View für das Feld Name **forschen** eingetragen. Abbildung 5.32 zeigt das Resultat.

Die Operation **vortragHalten** wird erstellt: Auswahl des Werkzeuges **Operation**, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **vortragHalten**. Abbildung 5.33 zeigt das Resultat.

Die Operation **mitarbeiterMotivieren** wird erstellt: Auswahl des Werkzeuges **Operation**, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **mitarbeiterMotivieren**. Abbildung 5.34 zeigt das Resultat.

Die Operation **prüft** wird erstellt: Auswahl des Werkzeuges **Operation**, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **prüft**. Nun muss noch der Parameter student angegeben werden. Dies geschieht mit Hilfe der Outline. Zunächst ist die Operation in der Zeichenfläche durch Klick auszuwählen. In der Outline erscheint nun die Operation. Hier wird durch Rechtsklick auf die Operation das Kontextmenü geöffnet und über ⟨`Create child`⟩→⟨`Parameter`⟩ der Parameter für die Operation erstellt. Durch Klick auf den neu erstellten Parameter in der Outline wird die zugehörige Properties-View geöffnet und dort für das Feld name **student** eingetragen. Weiter muss der Typ des Parameters, hier Student, angegeben werden. Durch Auswahl des Typs **Vorlesung** in der Property-View des gerade erzeugten Parameters wird dies umgesetzt. Abbildung 5.35 zeigt das Resultat.

Die Operation **publiziertDokument** wird erstellt: Auswahl des Werkzeuges **Ope-**

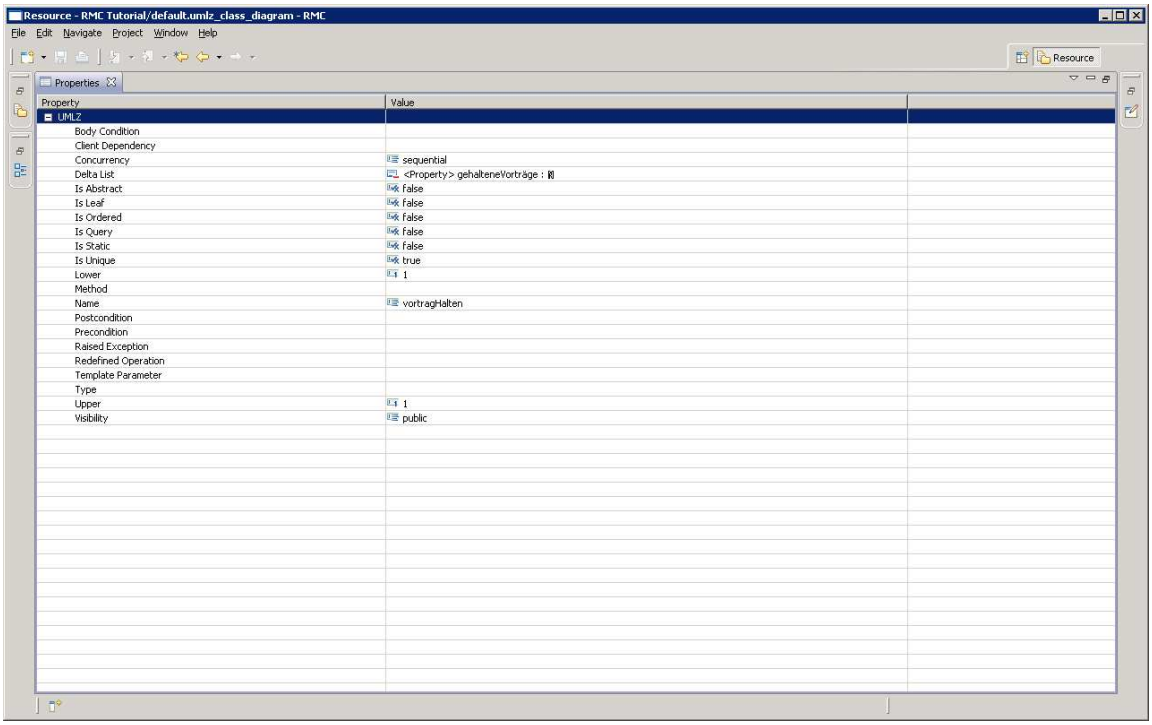


Abbildung 5.33: Tutorial: Eigenschaften der Methode vortragHalten

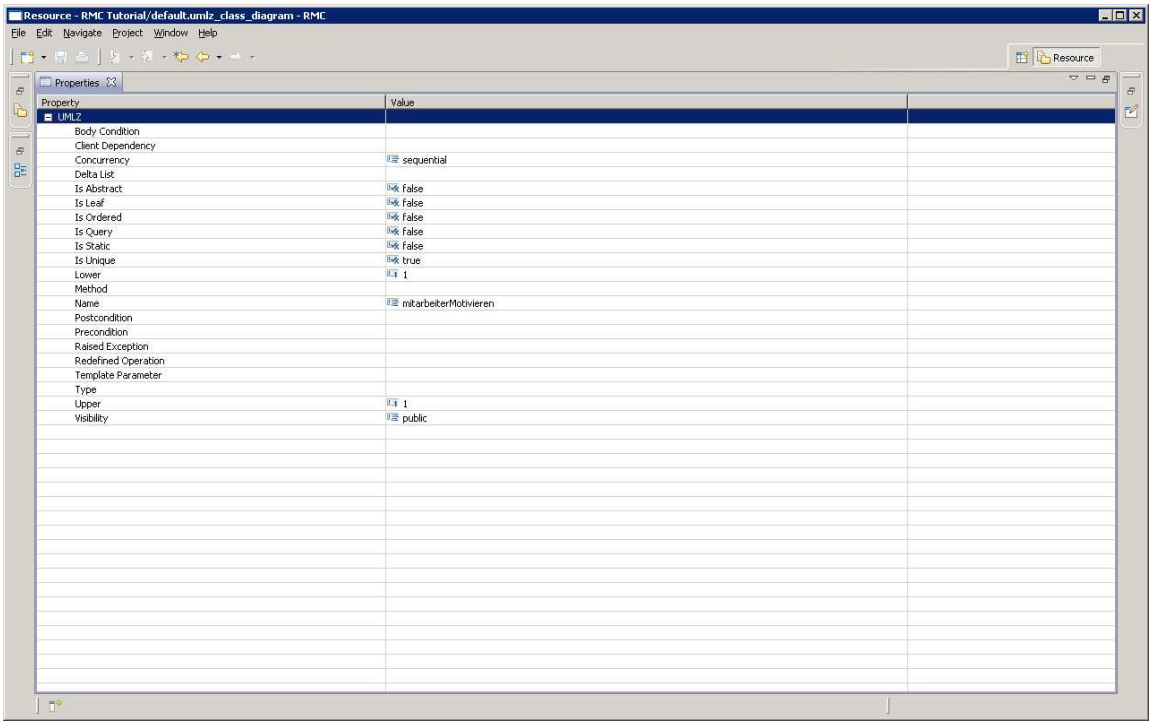


Abbildung 5.34: Tutorial: Eigenschaften der Methode mitarbeiterMotivieren

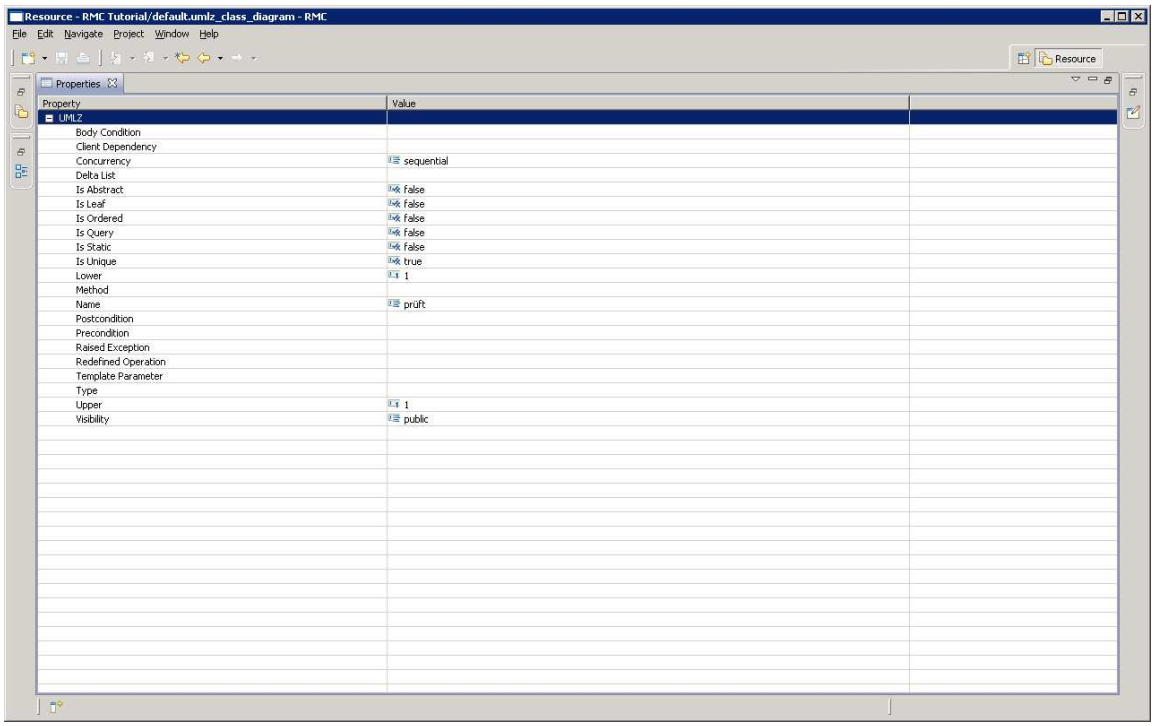


Abbildung 5.35: Tutorial: Eigenschaften der Methode prüft

ration, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **publiziertDokument**. Abbildung 5.36 zeigt das Resultat.

Die Operation **hältZentralübung** wird erstellt: Auswahl des Werkzeuges **Operation**, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **hältZentralübung**. Abbildung 5.37 zeigt das Resultat.

Die Operation **hältÜbung** wird erstellt: Auswahl des Werkzeuges **Operation**, Hinzufügen durch Klick auf die Klasse Professor und setzen des Namens **haeltUebung**. Abbildung 5.38 zeigt das Resultat.

Die Klasse Vorlesung: Durch erneute Wahl des Werkzeug **Class** und anschließendem Klick in die Zeichenfläche erscheint eine neue Klasse, in deren Eigenschaften der **Name Vorlesung** eingegeben wird. Abbildung 5.39 zeigt die resultierenden Eigenschaften.

Nun werden dieser Klasse noch vier Attribute hinzugefügt. Hierzu wird aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Vorlesung geklickt. Im erscheinenden Eingabefeld wird der Name des Attributes, in diesem Fall **zeit**, eingegeben. Abbildung 5.40 zeigt das Resultat.

Um das zweite Attribut zu erstellen, wird wieder aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Vorlesung geklickt. Im erscheinenden Eingabefeld wird der Name des Attributes, jetzt **fachgebiet**, eingegeben. Abbildung 5.41 zeigt das Resultat.

Das dritte Attribut wird erstellt, indem wieder aus der Palette das Werkzeug **Attribute** ausgewählt und anschließend in der Zeichenfläche auf die Klasse Vorlesung

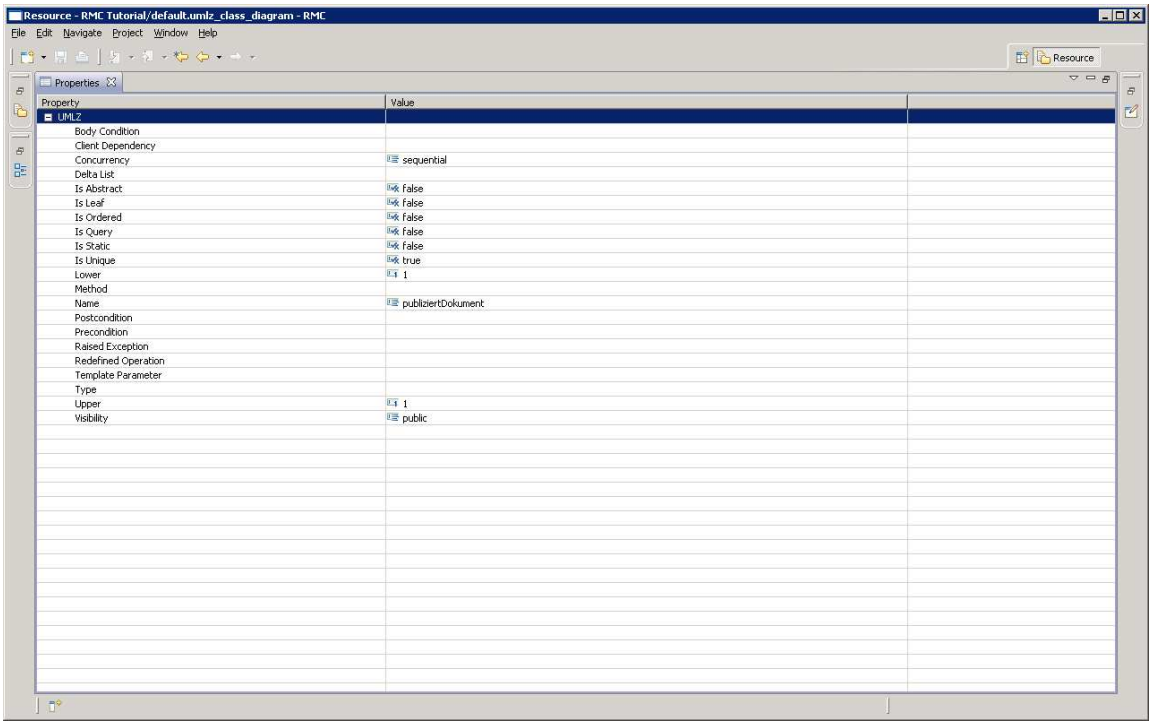


Abbildung 5.36: Tutorial: Eigenschaften der Methode publiziertDokument

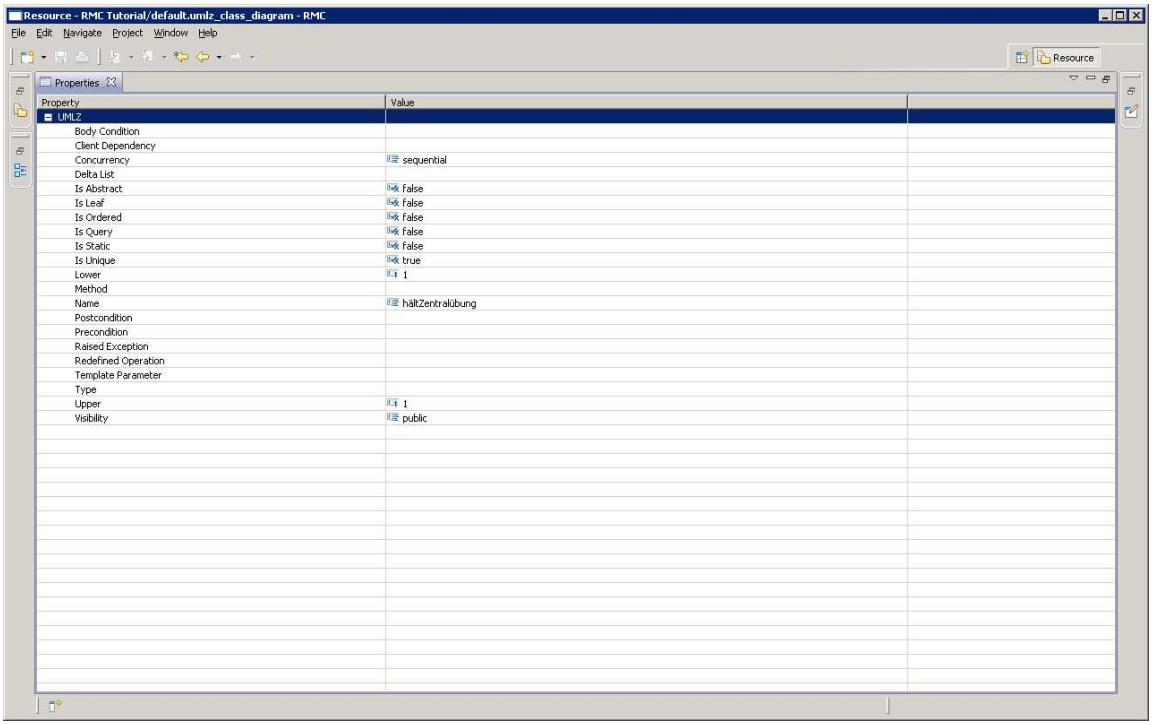


Abbildung 5.37: Tutorial: Eigenschaften der Methode hältZentralübung

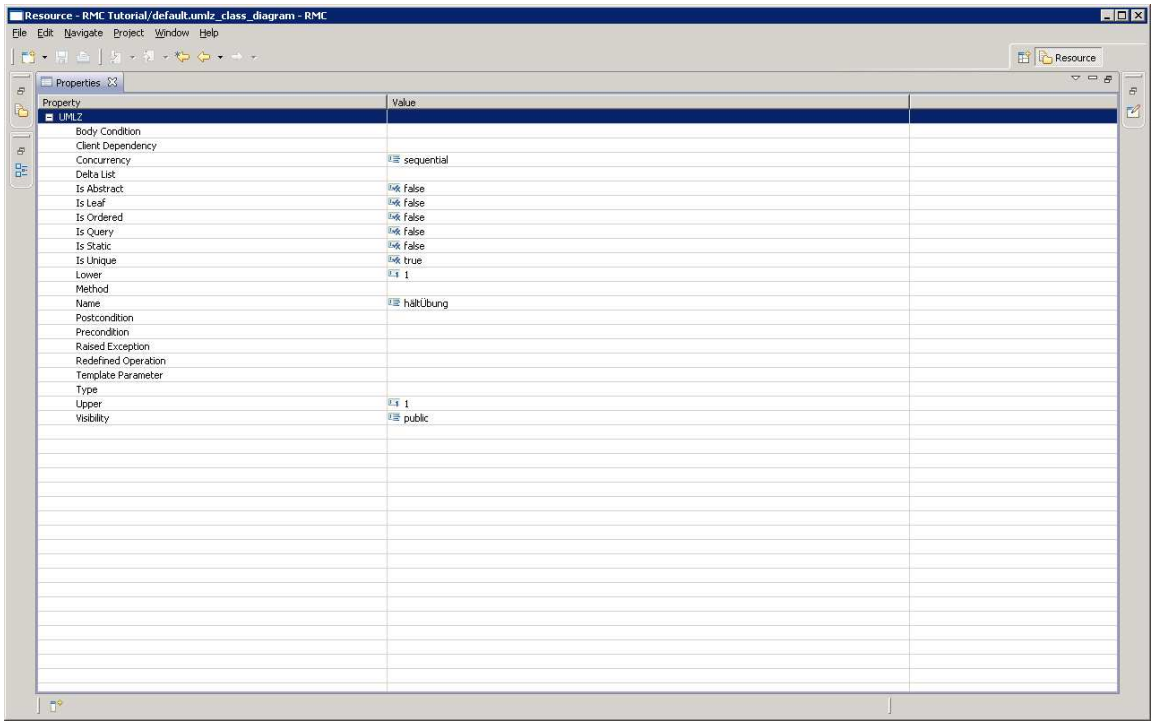


Abbildung 5.38: Tutorial: Eigenschaften der Methode hältÜbung

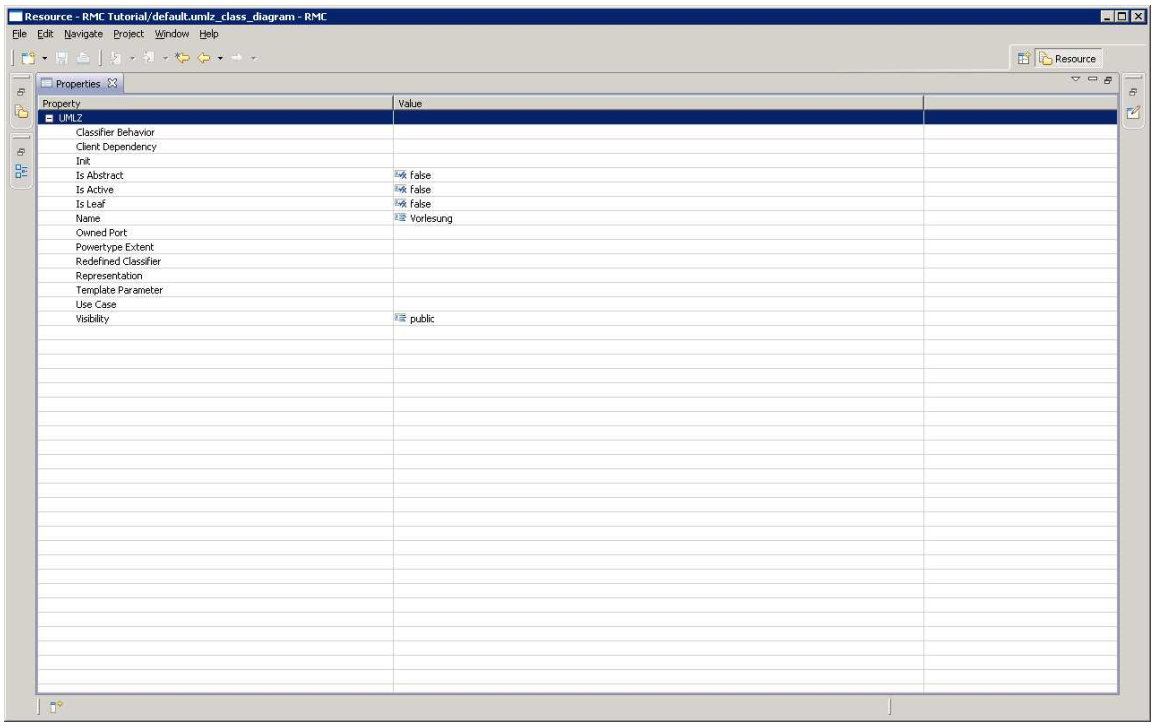


Abbildung 5.39: Tutorial: Eigenschaften der Klasse Vorlesung

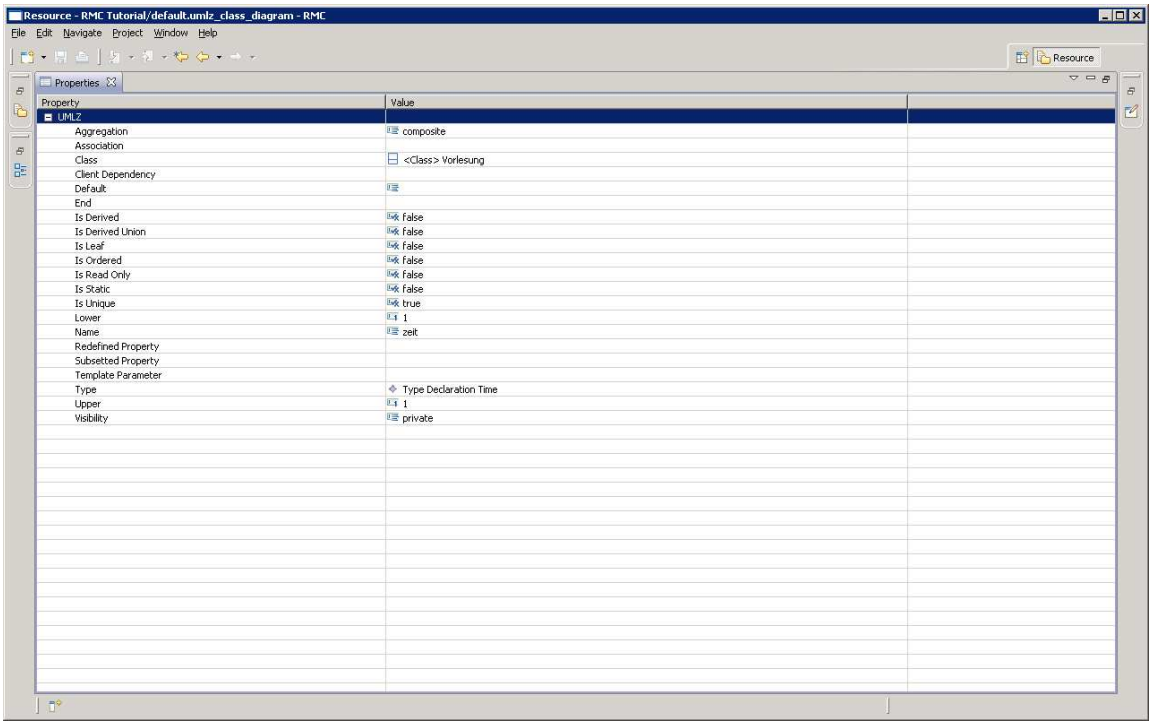


Abbildung 5.40: Tutorial: Eigenschaften des Attributes Zeit

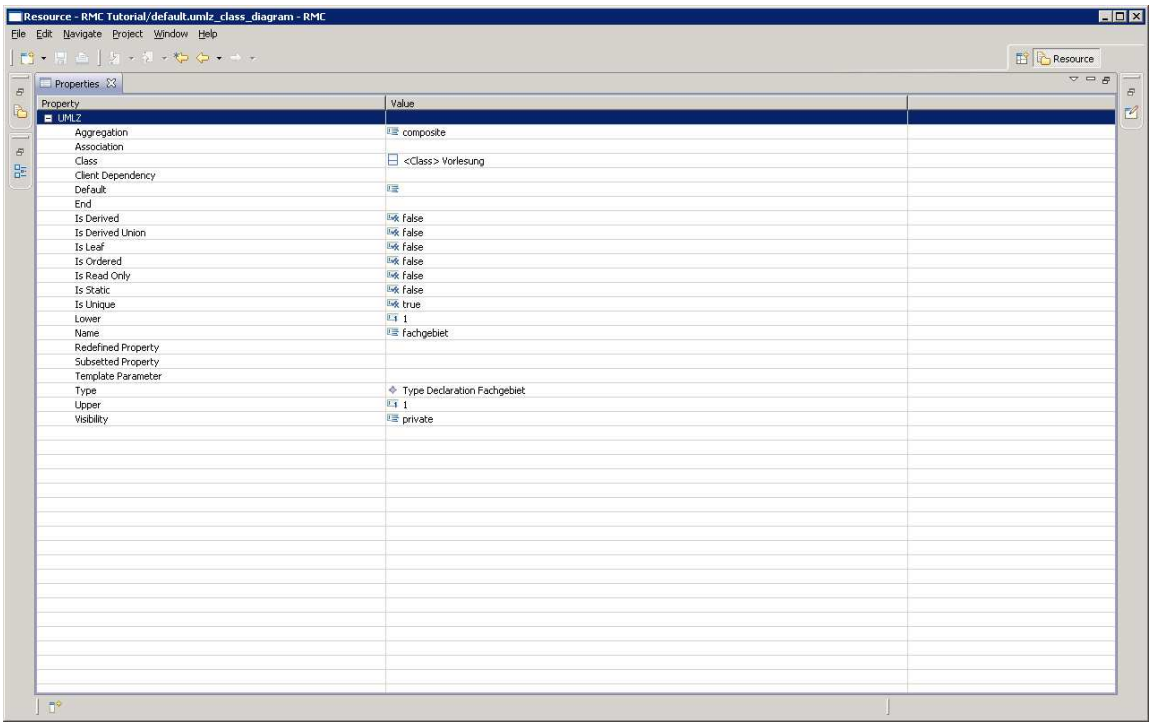


Abbildung 5.41: Tutorial: Eigenschaften des Attributes Fachgebiet

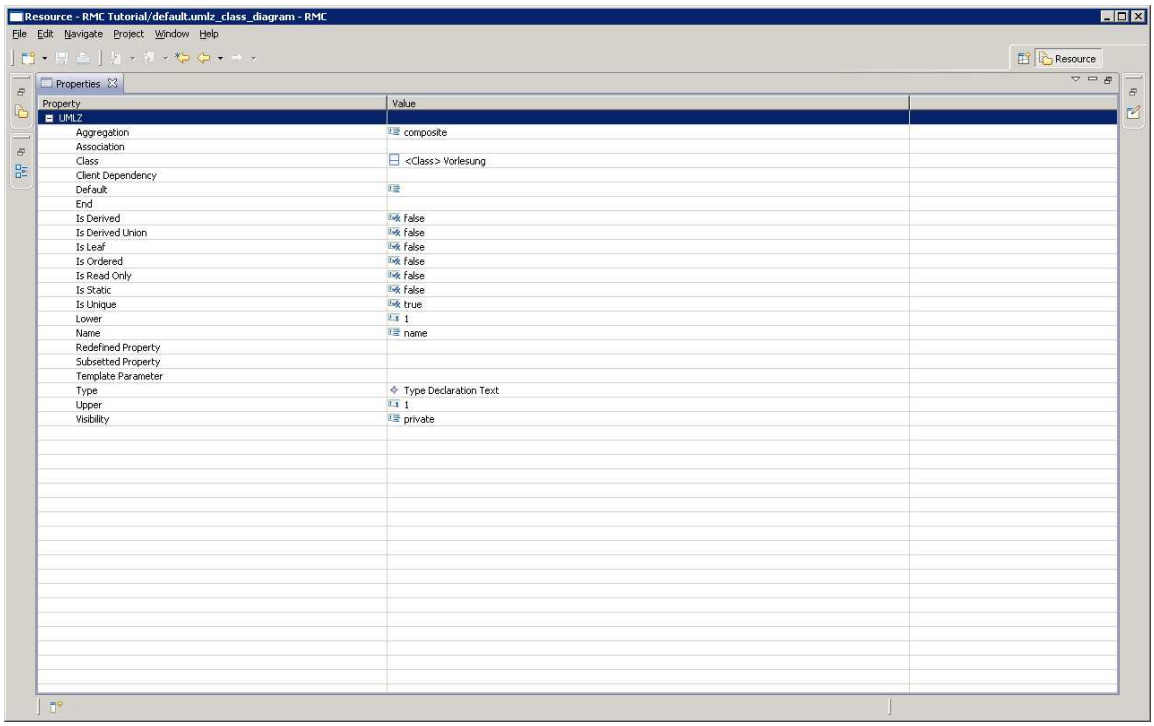


Abbildung 5.42: Tutorial: Eigenschaften des Attributes Name

geklickt. Im erscheinenden Eingabefeld wird der Name des Attributes, nun **name**, eingegeben. Abbildung 5.42 zeigt das Resultat.

Das letzte Attribut wird analog erstellt, indem mit dem Werkzeug **Attribute** auf die Klasse Vorlesung geklickt und als Name **raum** angegeben wird. Abbildung 5.43 zeigt das Resultat.

5.2.1.4 Beziehungen der Klassen modellieren

Im letzen Schritt werden die Beziehungen der Klassen modelliert. Dazu werden die Vererbungsbeziehungen und Assoziationen der Klassen angelegt. Abbildung 5.20 zeigt das Diagramm im Editor.

Vererbung Student: Um eine Vererbung von der Klasse Person auf die Klasse Student zu modellieren, wählt der Anwender aus der Palette das Werkzeug **Generalization** aus. Dann wird in der Zeichenfläche zunächst auf die Klasse Person und schließlich auf die Klasse Student geklickt. Abbildung 5.44 zeigt das Resultat.

Vererbung Professor: Um eine Vererbung von der Klasse Person auf die Klasse Professor zu modellieren, wählt der Anwender aus der Palette das Werkzeug **Generalization** aus. Dann wird in der Zeichenfläche zunächst auf die Klasse Person und schließlich auf die Klasse Professor geklickt. Abbildung 5.45 zeigt das Resultat.

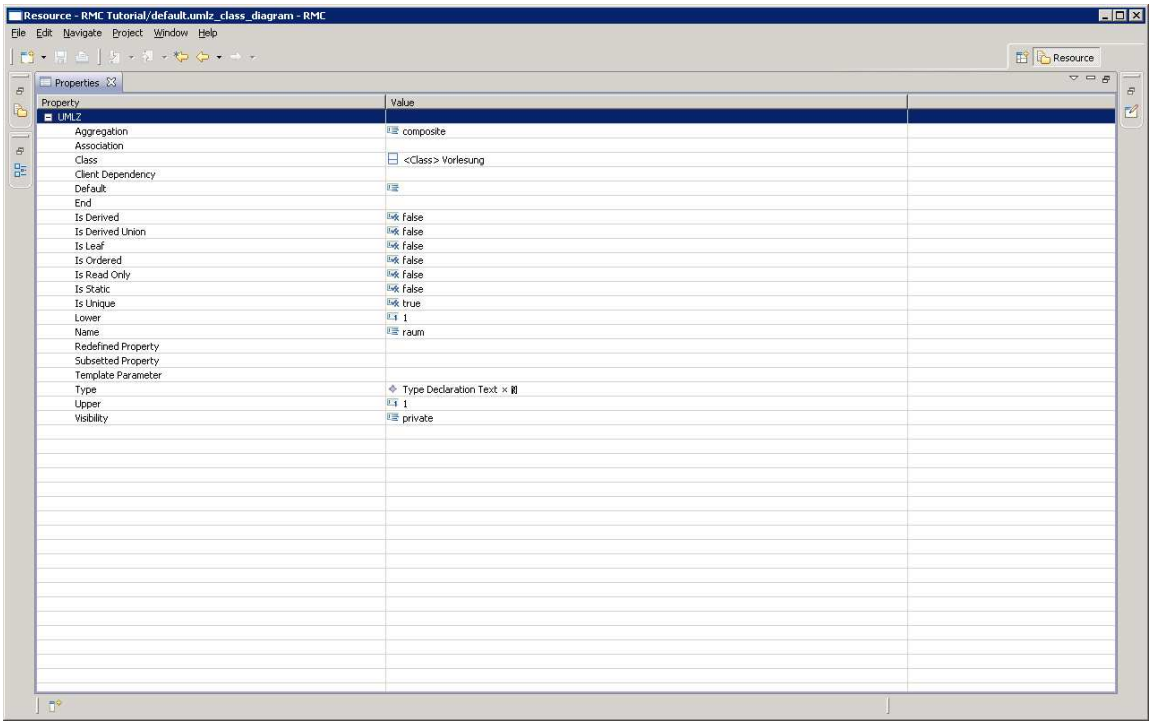


Abbildung 5.43: Tutorial: Eigenschaften des Attributes Raum

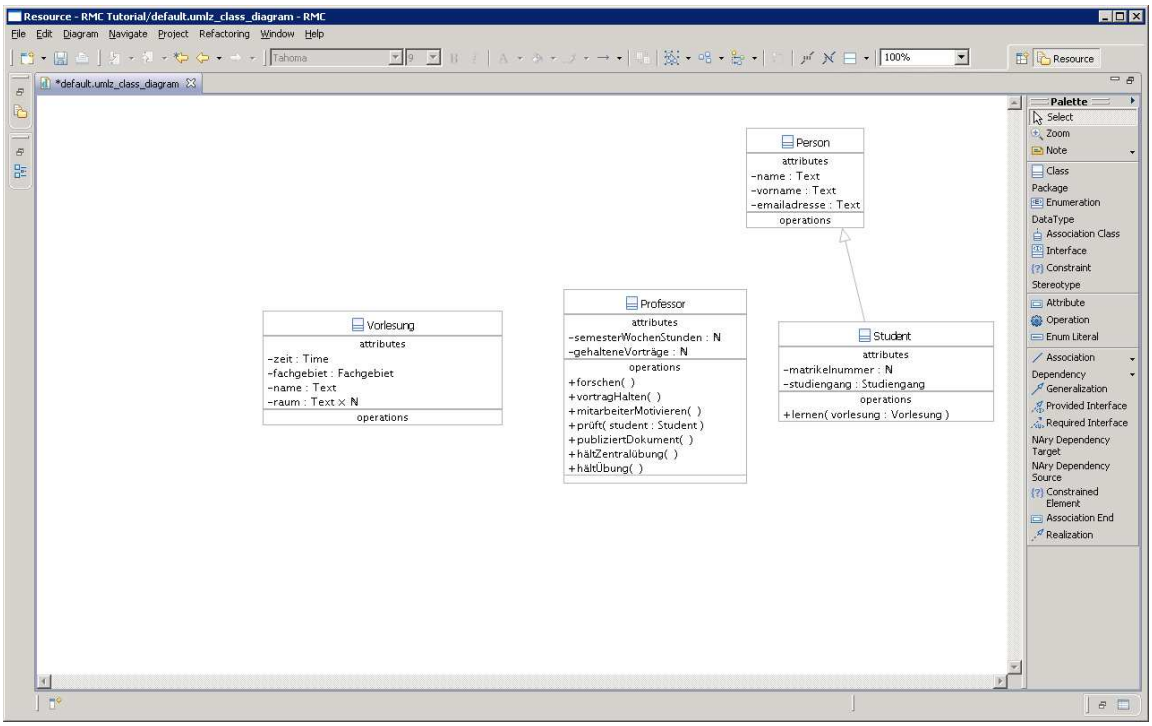


Abbildung 5.44: Tutorial: Generalisierung Student

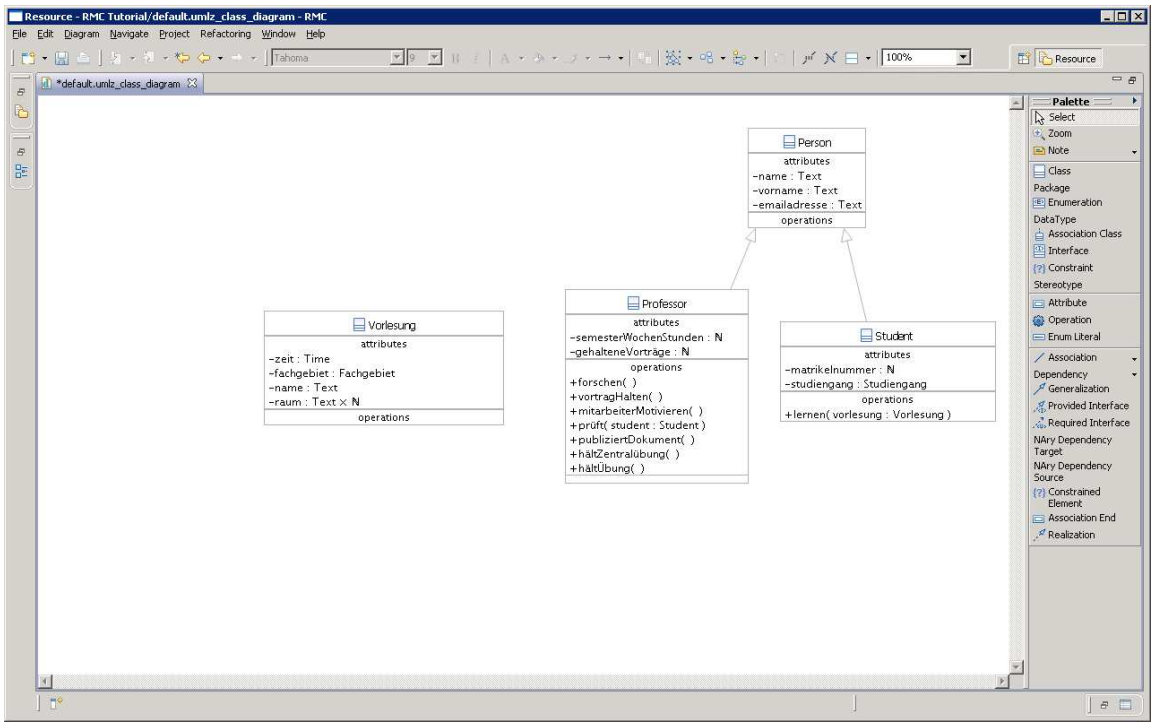


Abbildung 5.45: Tutorial: Generalisierung Professor

Assoziation hält: Um eine Assoziation **hält** zwischen den Klassen Professor und Vorlesung zu modellieren, wählt der Anwender aus der Palette das Werkzeug **Association** aus. Dann wird in der Zeichenfläche nacheinander auf die Klassen Professor und Vorlesung geklickt. Abbildung 5.46 zeigt das Resultat.

Assoziation nimmt teil: Um eine Assoziation **nimmt teil** zwischen den Klassen Student und Vorlesung zu modellieren, wählt der Anwender aus der Palette das Werkzeug **Association** aus. Dann wird in der Zeichenfläche nacheinander auf die Klassen Student und Vorlesung geklickt. Abbildung 5.47 zeigt das Resultat.

Das UML/Z-Modell ist nun vollständig und entspricht dem dargestellten Modell aus Abbildung 5.1.

5.2.2 Modellqualitätsplan

Nachdem ein UML/Z-Modell angelegt wurde, kann mit der Erstellung eines Modellqualitätsplans (MQP) begonnen werden. Dieses Kapitel führt den Benutzer Schritt für Schritt durch das Anlegen der einzelnen Elemente eines Modellqualitätsplans. Resultat wird ein kompletter MQP sein, mit dessen Hilfe eine Messung auf dem zuvor definierten UML/Z-Modell durchgeführt werden kann.

Das Kapitel gliedert sich in folgende Teilbereiche:

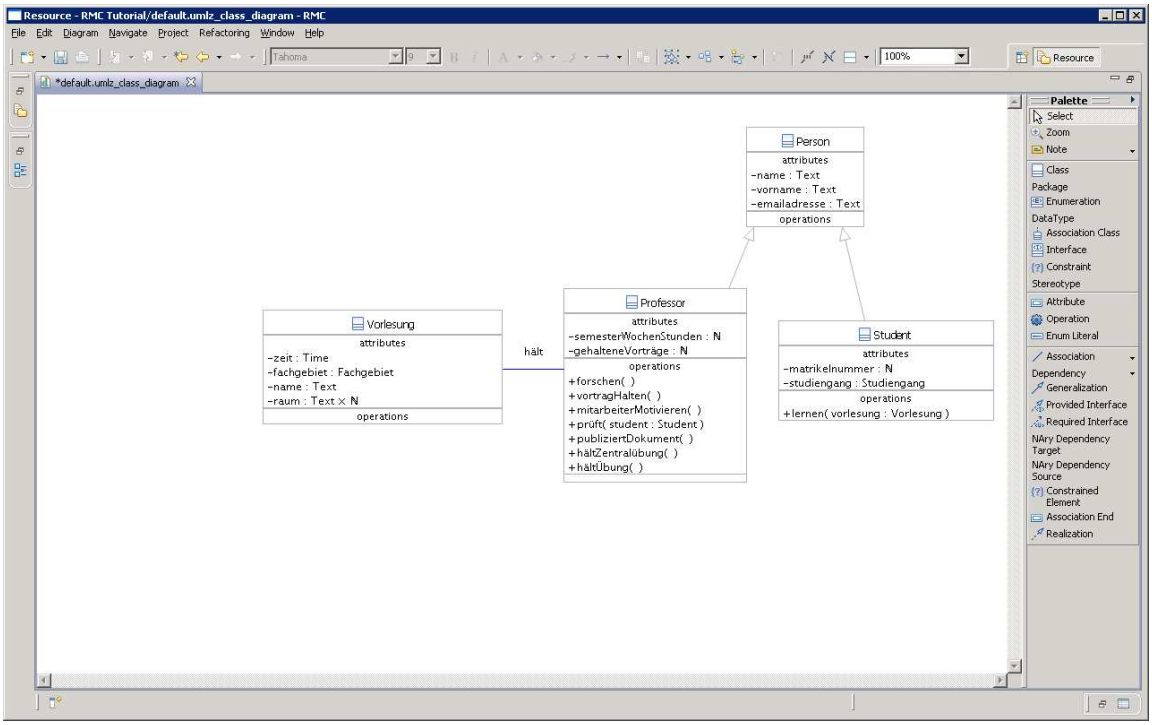


Abbildung 5.46: Tutorial: Assoziation hält

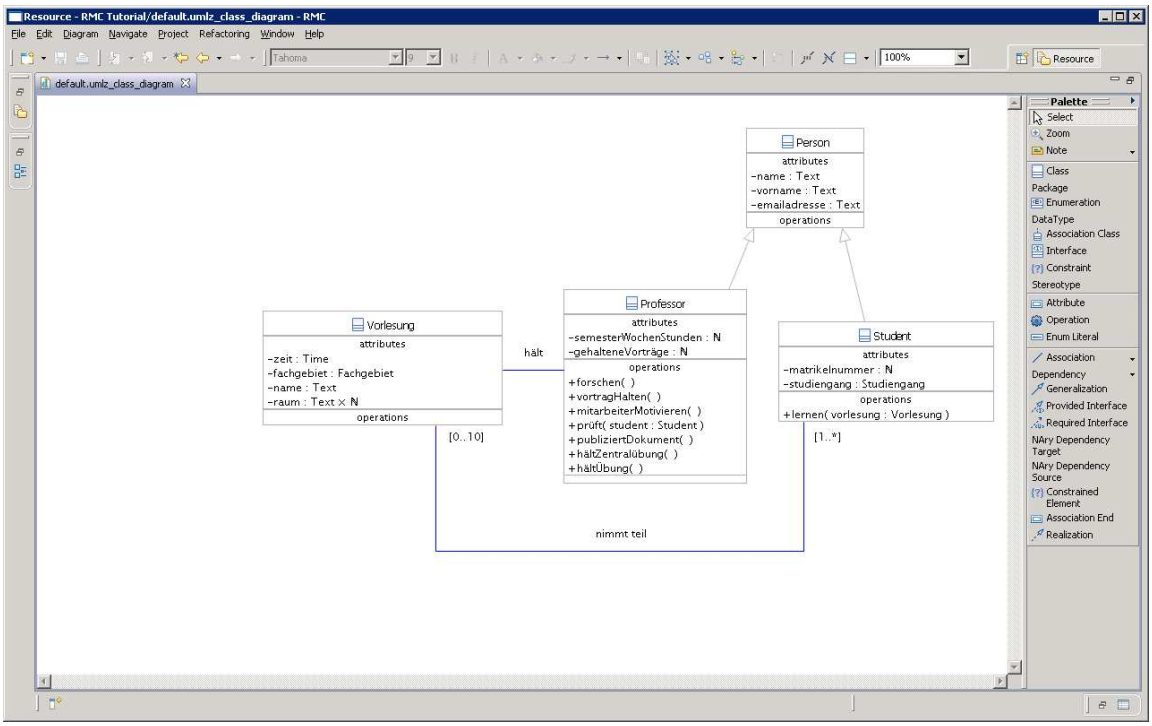


Abbildung 5.47: Tutorial: Assoziation nimmt teil

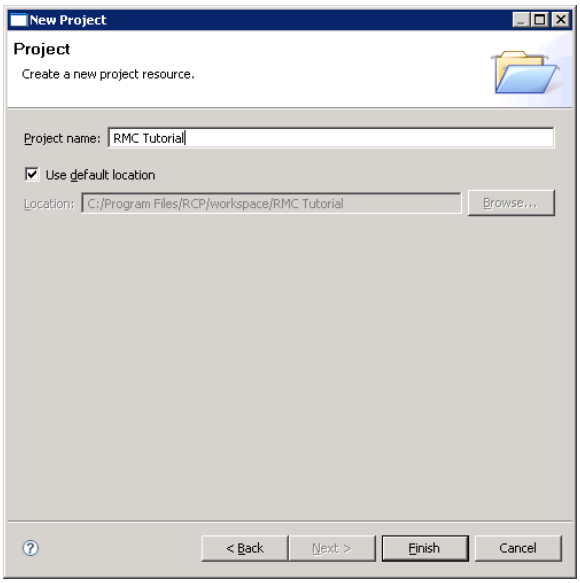


Abbildung 5.48: Neues allgemeines Projekt anlegen

- Einen neuen Modellqualitätsplan anlegen
- Ein neues Kontextmodell erstellen
- Ein neues Informationsbedürfnismodell erstellen
- Ein neues Qualitätsmodell erstellen
- Ein neues Messmodell erstellen
- Das Präsentationsmodell verwenden

5.2.2.1 Einen neuen Modellqualitätsplan anlegen

Als erstes wird ein neues allgemeines Projekt benötigt: ⟨`File`⟩→⟨`New`⟩→⟨`General`⟩. Im erscheinenden Dialog (Abbildung 5.48) wird der Name des Projektes (RMC Tutorial) eingegeben und mit [Next] bestätigt.

Nachdem das neue Projekt angelegt wurde, kann ein neuer **Modellqualitätsplan** angelegt werden: ⟨`File`⟩→⟨`New`⟩→⟨`Other`⟩→⟨`RMC`⟩→⟨`Model Quality Plan`⟩ (siehe Abbildung 5.49).

Nachdem der Anwender „Model Quality Plan“ ausgewählt und mit [Next] bestätigt hat, erscheint ein weiterer Dialog, in dem der Name des Modellqualitätsplans eingegeben werden muss. Abbildung 5.50 zeigt diesen Dialog. Der Anwender gibt **RMC_MQP_Tutorial** als Dateinamen ein, wählt den Ordner des zuvor angelegten allgemeinen Projektes (RMC Tutorial) und bestätigt mit [Next]. Es wird daraufhin eine neuer Modellqualitätsplan mit dem Namen **RMC_MQP_Tutorial** angelegt.

In dem sich daraufhin öffnenden Dialog gibt der Anwender den Modellqualitätsplan Namen (**model quality plan name**) und die Version (**version**) ein. Anschließend wählt er das

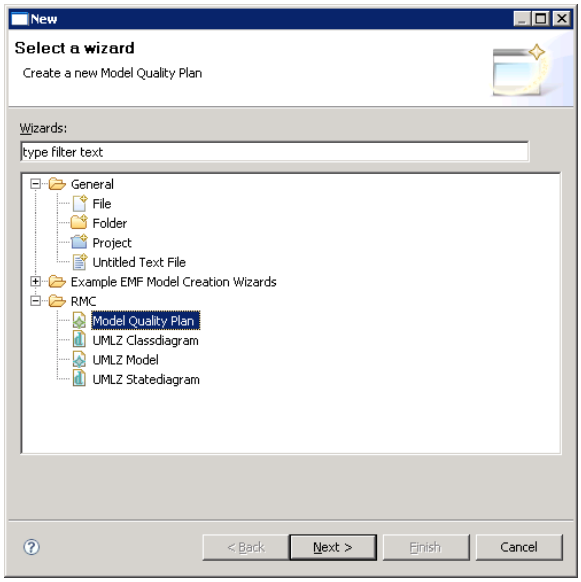


Abbildung 5.49: Neuen Modellqualitätsplan anlegen

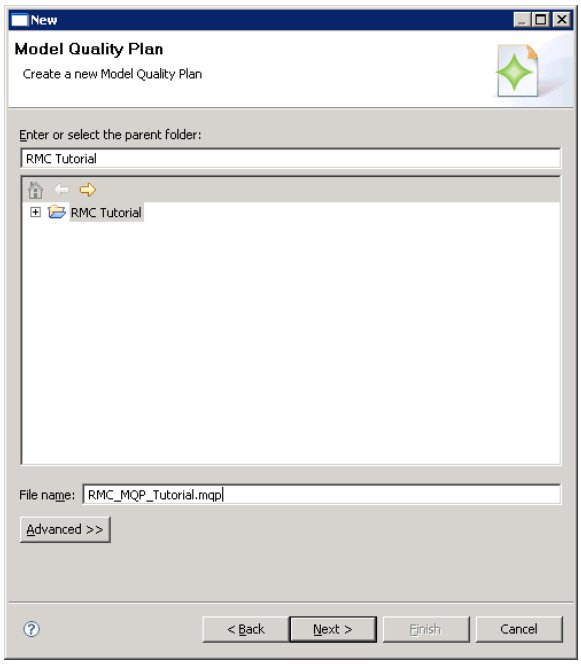


Abbildung 5.50: Modellqualitätsplan anlegen

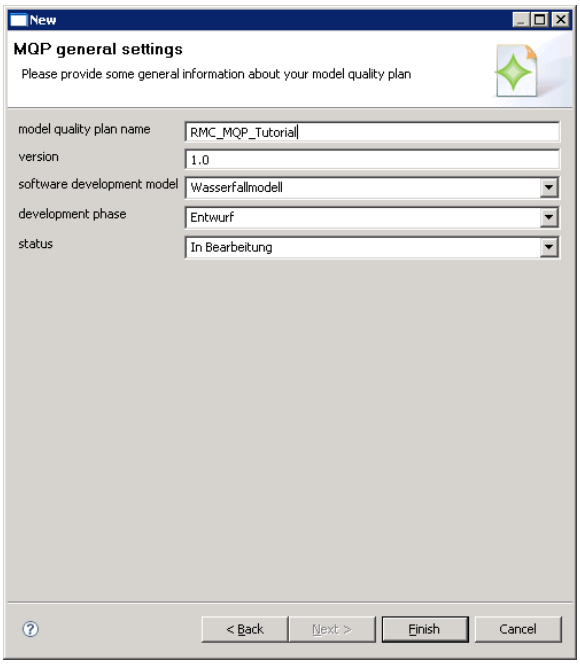


Abbildung 5.51: Modellqualitätsplan - grundlegende Einstellungen

verwendete Vorgehensmodell (**software development model**) und die aktuelle Entwicklungsphase (**development phase**) aus. In unserem Beispiel ist der Name **MQP Tutorial**, die Version **1.0**, das Vorgehensmodell **Wasserfallmodell** und die Entwicklungsphase **Entwurf**. Im letzten Schritt dieses Dialoges gibt der Anwender den Status (**status**) (hier **In Bearbeitung**) des Modellqualitätsplans ein und bestätigt mit [Next]. Abbildung 5.50 zeigt den ausgefüllten Dialog.

Das Modelqualitätsplan-Projekt ist nun erfolgreich angelegt und es kann mit der Erstellung des Kontextmodells begonnen werden.

5.2.2.2 Ein neues Kontextmodell erstellen

Nachdem der Anwender den Moderator des MQPs eingegeben hat (siehe Abbildung 5.52, oben rechts), fügt er im **„Autoren-Bereich“** mit [Add Author] einen Autor dem MQP hinzu. Der in dem sich öffnenden Dialog eingegebene Autor kann mit [Edit Author] und [Delete Author] bearbeitet bzw. gelöscht werden. Jeder MQP hat immer genau einen Moderator und mindestens einen Autor, wobei der Moderator auch einer der Autoren sein kann.

Im nächsten Schritt gibt der Anwender zwei Rollen **Architekt** und **Entwickler** [Add Role] ein. Der Name sowie eine Beschreibung einer Rolle kann über einen Dialog eingegeben werden. Mit [Edit Role] und [Delete Role] kann eine Rolle bearbeitet bzw. gelöscht werden.

Sobald die beiden Rollen eingegeben wurden, können in „Artefakts-Bereich“ Artefakte über [Add Artefact] hinzugefügt werden. In dem sich öffnenden Dialog können Artefakte vom

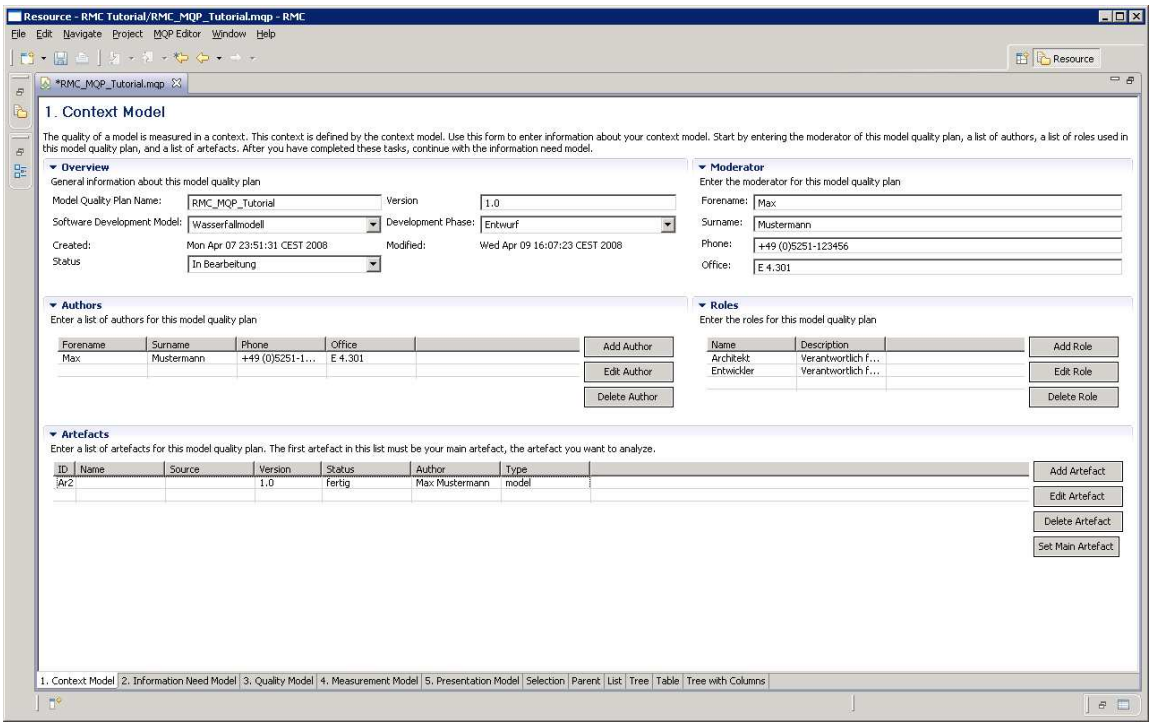

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
*RMC_MQP_Tutorial.mqp
1. Context Model
The quality of a model is measured in a context. This context is defined by the context model. Use this form to enter information about your context model. Start by entering the moderator of this model quality plan, a list of authors, a list of roles used in this model quality plan, and a list of artefacts. After you have completed these tasks, continue with the information need model.
Overview
General information about this model quality plan
Model Quality Plan Name: RMC_MQP_Tutorial
Version: 1.0
Software Development Model: Wasserfallmodell
Development Phase: Entwurf
Created: Mon Apr 07 23:51:31 CEST 2008
Modified: Wed Apr 09 16:07:23 CEST 2008
Status: In Bearbeitung
Moderator
Enter the moderator for this model quality plan
Forename: Max
Surname: Mustermann
Phone: +49 (0)5251-123456
Office: E 4.301
Authors
Enter a list of authors for this model quality plan
Forename Surname Phone Office
Max Mustermann +49 (0)5251-1... E 4.301
Add Author
Edit Author
Delete Author
Roles
Enter the roles for this model quality plan
Name Description
Architekt Verantwortlich f...
Entwickler Verantwortlich f...
Add Role
Edit Role
Delete Role
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
Ar2 1.0 fertig Max Mustermann model
Add Artefact
Edit Artefact
Delete Artefact
Set Main Artefact
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns

Abbildung 5.52: Kontextmodell anlegen

Typ **software model**, **software requirements specification** oder **software code** eingegeben werden. Beim erstmaligen Anlegen eines Artefakts ist **software model** vorausgewählt, da jeder MQP ein Softwaremodell, auf welches sich der MQP bezieht, beinhalten muss.

Nachdem eine ID (Id), die Entwicklungsphase (Phase), die Version (Version) und der Status (Status) für das Artefakt eingegeben bzw. ausgewählt wurden, kann mit [Add Author] ein Autor für dieses Artefakt eingegeben werden. Ein für dieses Beispiel ausgefüllter Dialog ist in Abbildung 5.53 abgebildet.

Anschließend muss der Anwender die Modellierungssprache (Modelling Language), den Modellierungszweck (Model Purpose), die Modellkategorie (Model Category) und die Modell-Methodik (Model Methodology) eingeben. Die Eingabe des Artefact bestätigt der Anwender mit [Save Changes].

5.2.2.3 Ein neues Informationsbedürfnismodell erstellen

Im Informationsbedürfnismodell gibt der Anwender zunächst drei Ziele [add Goals] ein. In diesem Beispiel sind das die Ziele:

- Implementierbarkeit bestimmen
- semantische Vollständigkeit

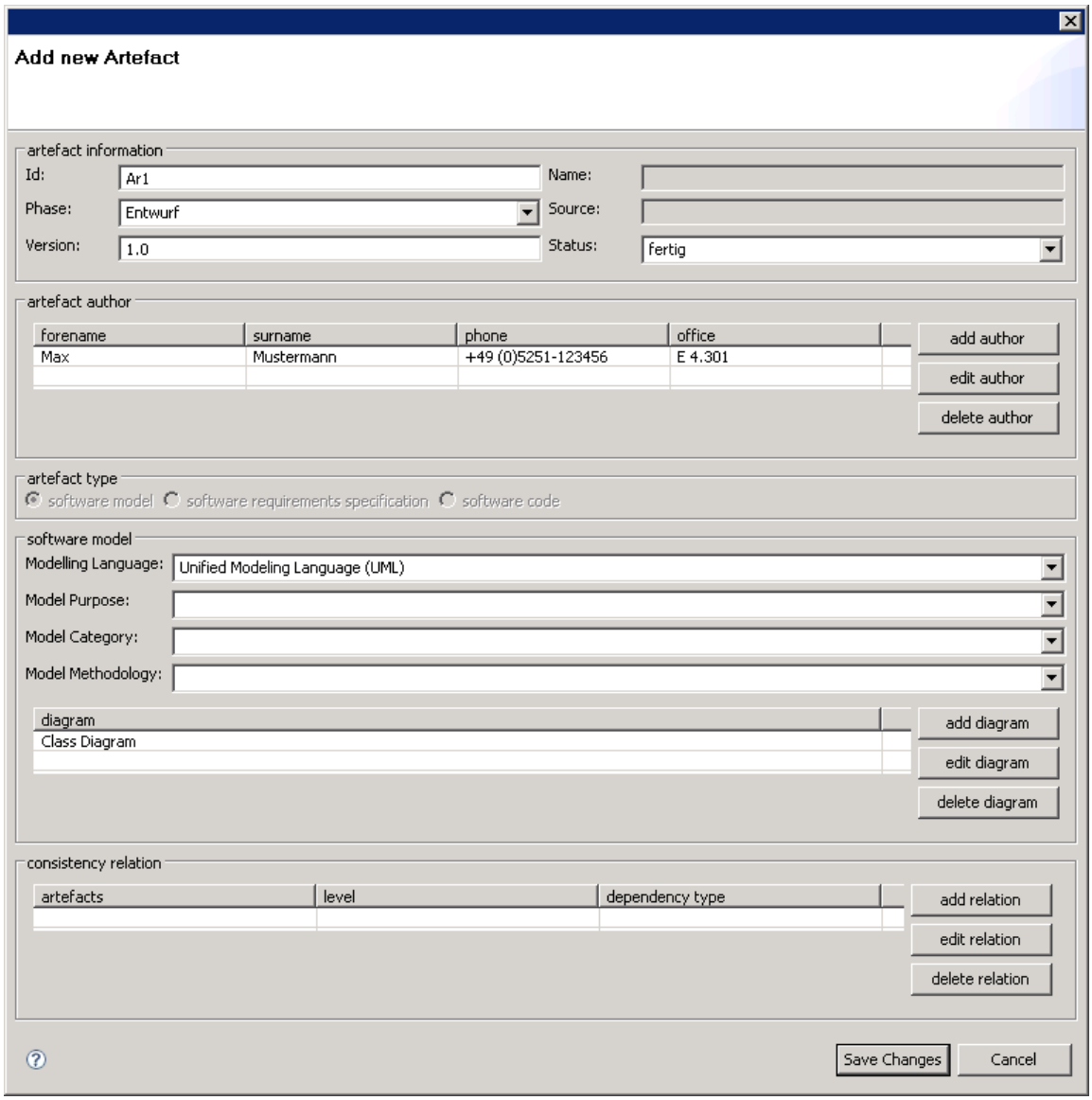


Abbildung 5.53: Artefakt hinzufügen

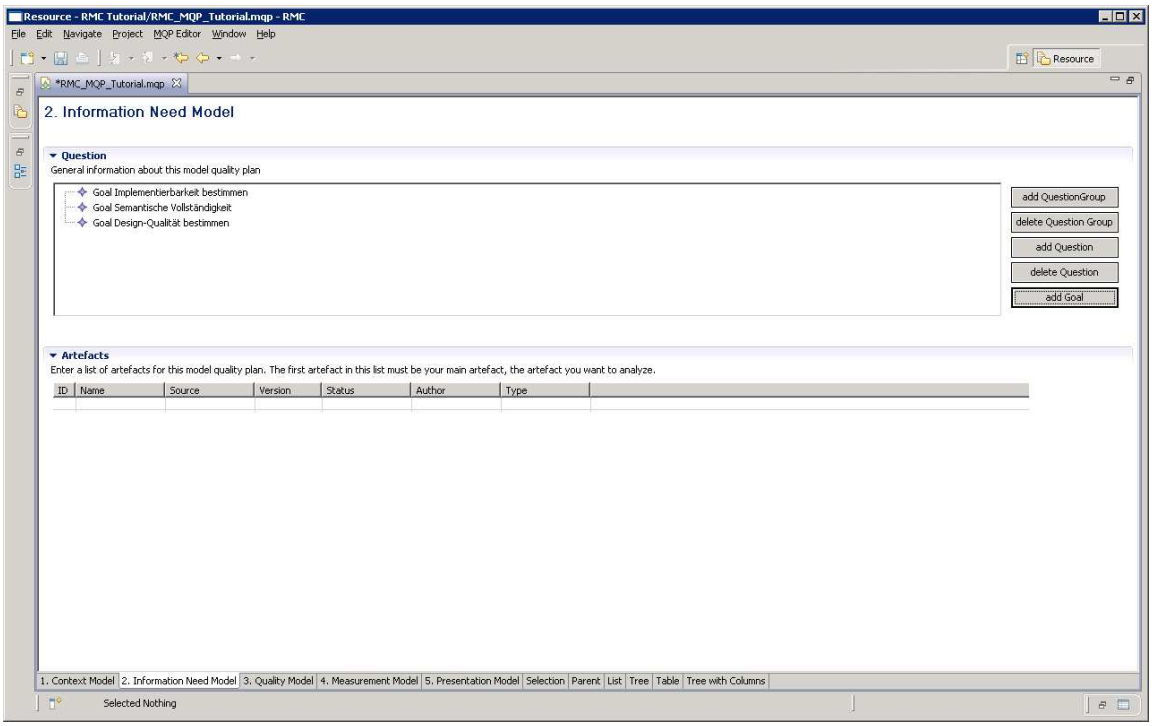

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
Resource
*RMC_MQP_Tutorial.mqp
2. Information Need Model
Question
General information about this model quality plan
Goal Implementierbarkeit bestimmen
Goal Semantische Vollständigkeit
Goal Design-Qualität bestimmen
add QuestionGroup
delete Question Group
add Question
delete Question
add Goal
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns
Selected Nothing

Abbildung 5.54: Informationsbedürfnismodell - Ziele

- Design-Qualität bestimmen

Beim Hinzufügen der Ziele ordnet er jedem Ziel die Rolle zu, die ein Interesse an dem entsprechenden Ziel hat (siehe Abbildung 5.55). Dies geschieht über die Properties des jeweiligen Ziels. Die drei Ziele werden wie folgt zugeordnet:

- Implementierbarkeit bestimmen → Entwickler
- semantische Vollständigkeit → Architekt
- Design-Qualität bestimmen → Architekt

Anschließend gibt der Anwender fünf Fragen ein und ordnet sie den Zielen per Drag-and-Drop zu. Die Fragen in diesem Beispiel sind:

- Wird Mehrfachvererbung eingesetzt?
- Sind alle Operationen vollständig spezifiziert?
- Wird Vererbung richtig eingesetzt?
- Gibt es Klassen mit zu vielen Methoden?
- Gibt es Klassen mit einem zu langen Namen?

Das fertig ausgefüllte Informationsbedürfnis ist in Abbildung 5.56 abgebildet.

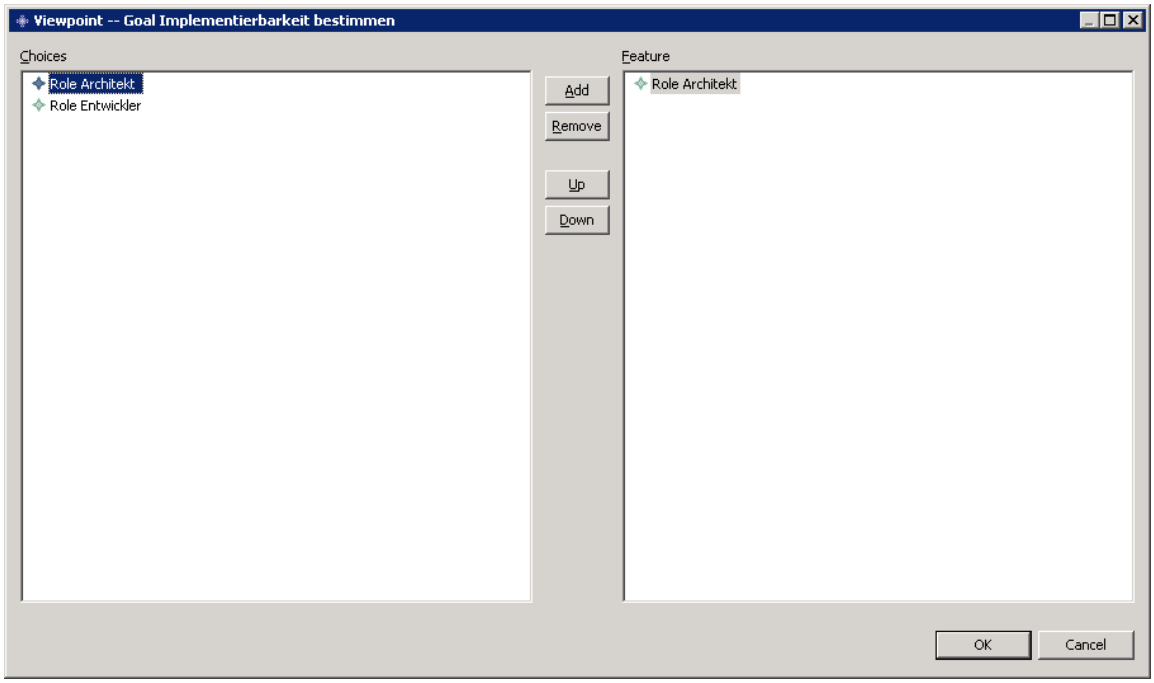


Abbildung 5.55: Zuordnung Ziele zu Rollen

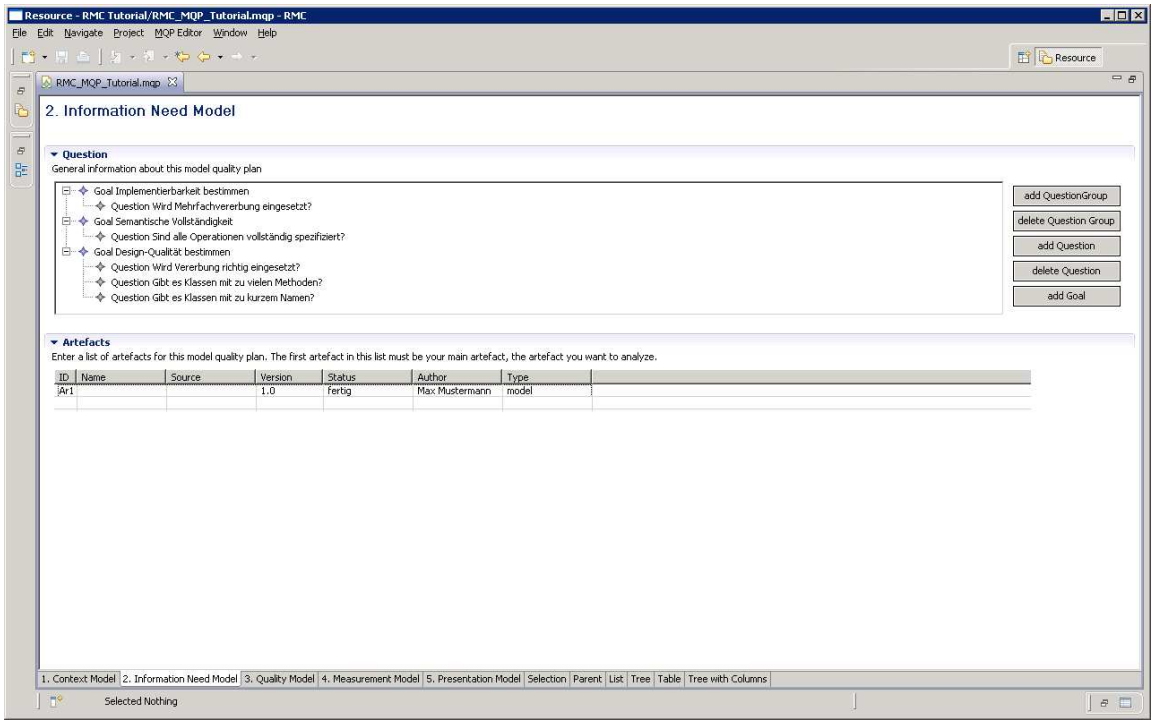

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
Resource
RMC_MQP_Tutorial.mqp
2. Information Need Model
Question
General information about this model quality plan
Goal Implementierbarkeit bestimmen
Question Wird Mehrfachvererbung eingesetzt?
Goal Semantische Vollständigkeit
Question Sind alle Operationen vollständig spezifiziert?
Goal Design-Qualität bestimmen
Question Wird Vererbung richtig eingesetzt?
Question Gibt es Klassen mit zu vielen Methoden?
Question Gibt es Klassen mit zu kurzem Namen?
add QuestionGroup
delete Question Group
add Question
delete Question
add Goal
Artefacts
Enter a list of artefacts for this model quality plan. The first artefact in this list must be your main artefact, the artefact you want to analyze.
ID Name Source Version Status Author Type
Ar1 1.0 fertig Max Mustermann model
1. Context Model
2. Information Need Model
3. Quality Model
4. Measurement Model
5. Presentation Model
Selection Parent List Tree Table Tree with Columns
Selected Nothing

Abbildung 5.56: Informationsbedürfnismodell

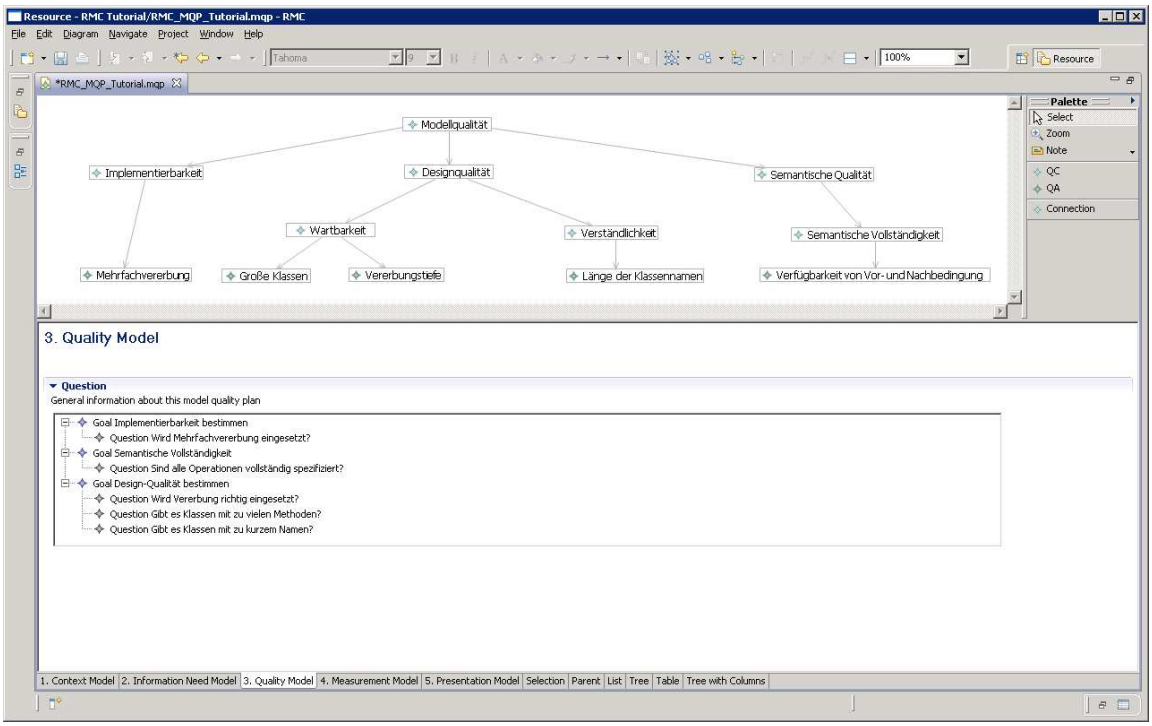


Abbildung 5.57: Qualitätsmodell

5.2.2.4 Ein neues Qualitätsmodell erstellen

Im Qualitätsmodell legt der Anwender zunächst im oberen Bereich die Qualitätscharakteristiken und Qualitätsattribute fest (siehe Abbildung 5.57). Dabei ordnet er jeder Qualitätscharakteristik ein Ziel über die Properties zu. Das Qualitätsmodell für dieses Beispiel ist in Abbildung 5.57 abgebildet.

Nachdem das Qualitätsmodell aufgebaut wurde, ordnet der Anwender jedem Qualitätsattribut eine Frage ebenfalls über die Properties zu (siehe Abbildung 5.58).

Das fertige Qualitätsmodell ist in Abbildung 5.59 abgebildet.

5.2.2.5 Ein neues Messmodell erstellen

Um nun eine Messung durchführen zu können, wechselt der Anwender in das Messmodell. Dort ordnet er den aufgelisteten Qualitätsattributen Messungen per Drag - and - Drop zu. Abbildung 5.60 zeigt das Messmodell und die Zuordnung der ersten vier Qualitätsattribute.

Bei diesen ersten vier Qualitätsattributen ist die Zuordnung leicht möglich, wohingegen für das fünfte Qualitätsattribut keine geeignete Messung existiert. Also muss der Anwender zunächst eine geeignete Messung mit [add measure] eingeben. Abbildung 5.61 zeigt den Dialog und die eingegebenen Messung.

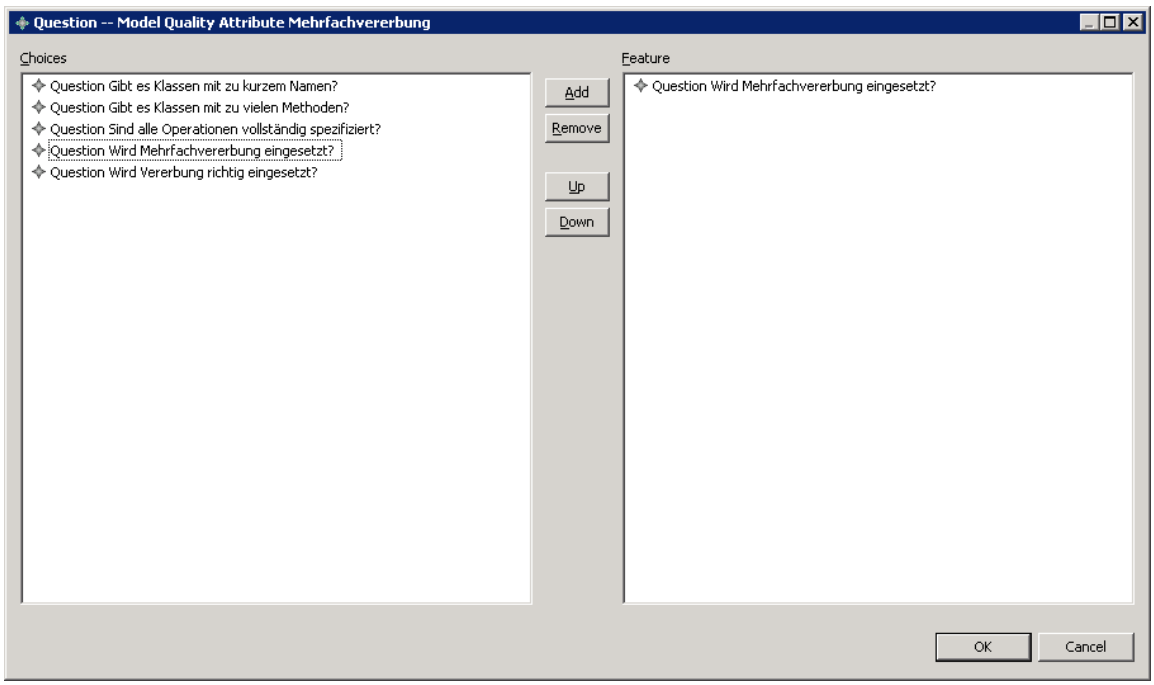


Abbildung 5.58: Qualitätsmodell - Zuordnung von Fragen

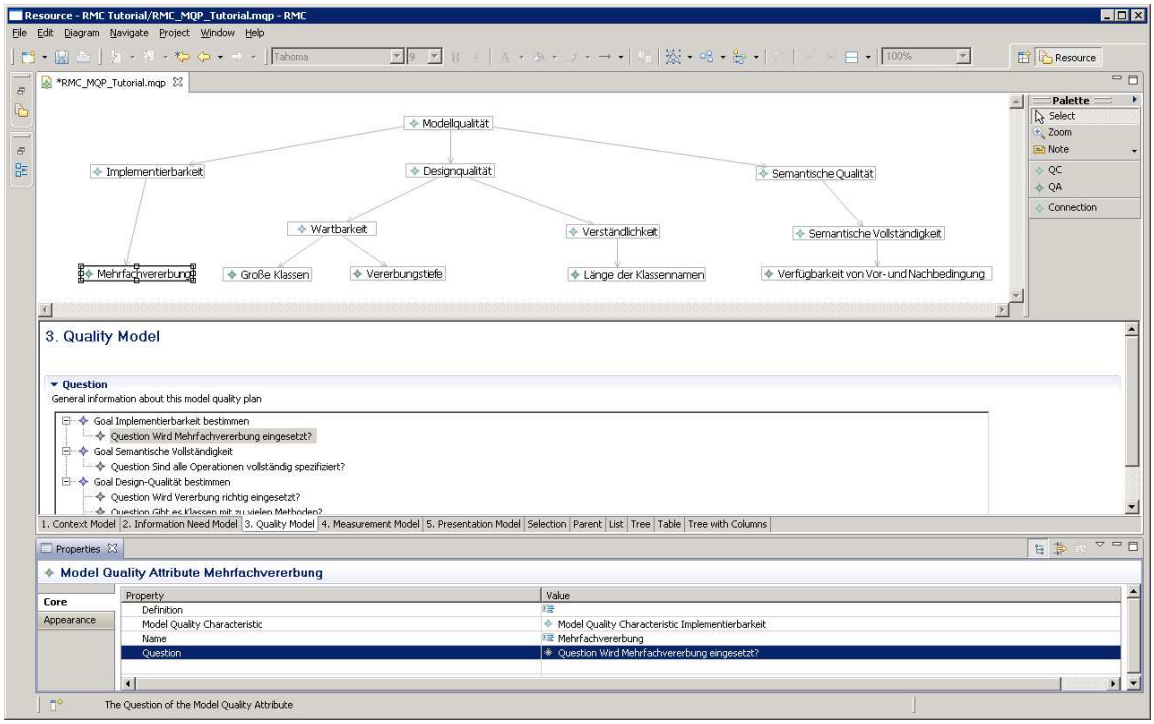


Abbildung 5.59: Qualitätsmodell

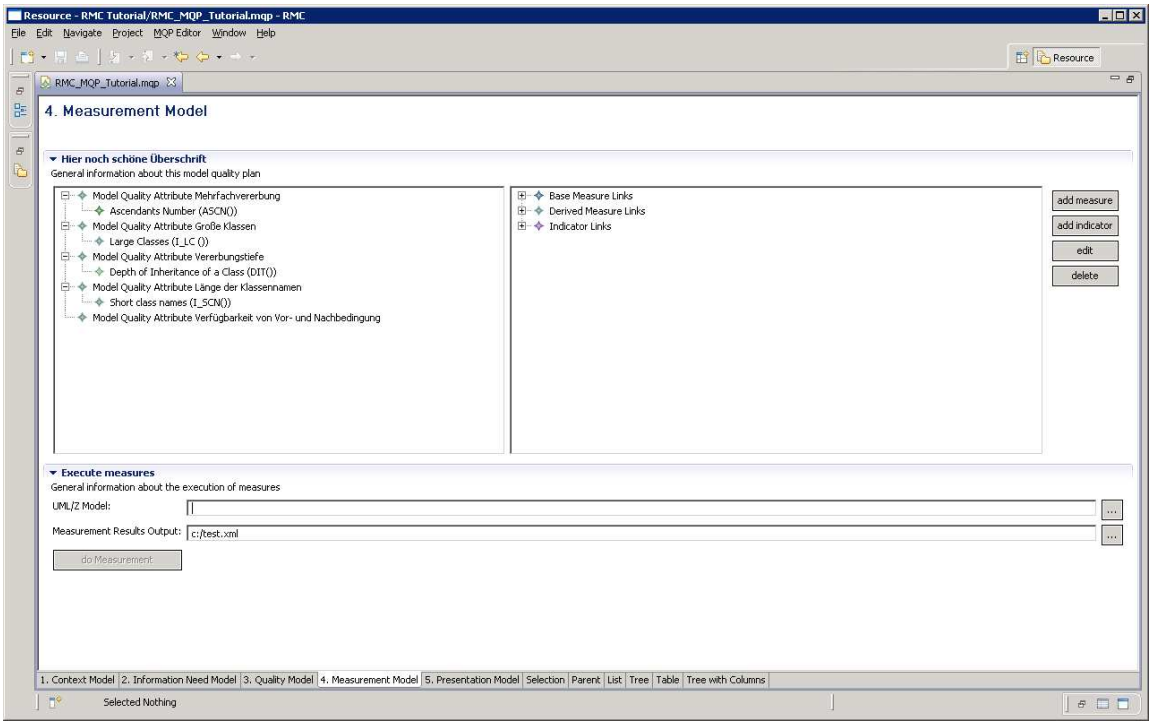


Abbildung 5.60: Zuordnung der Qualitätsattribute im Messmodell

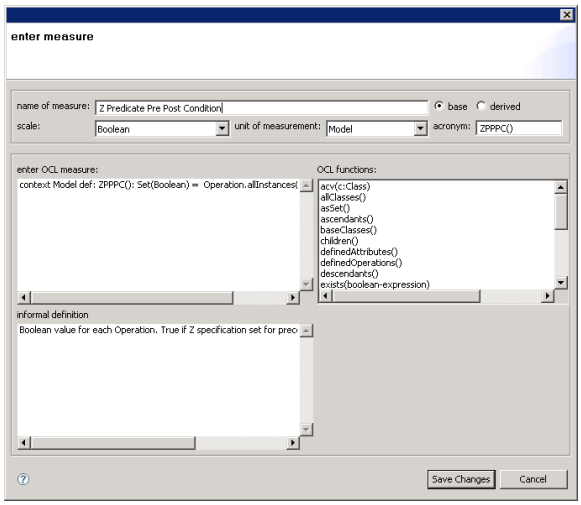


Abbildung 5.61: Eingabe einer neuen Messung

Zunächst wird der Name der neu anzulegenden Messung eingeben. Dieser Name sollte so gewählt werden, dass er eine grobe Beschreibung der Messung beinhaltet.

Im nächsten Schritt wird definiert, ob die Messung eine **base measure** oder eine Derived Measure darstellt. Base measures verwenden nur Hilfsfunktionen und Standard-OCL-Ausdrücke, während **derived measures** auch bereits definierte base und derived measures wiederverwenden. Über die Auswahlbox **scale** wird die Skala angegeben, die auf den Ergebniswert der Messung anzuwenden ist. Weiter wird in **unit of measurement** ein Modellelement-Typ angegeben, auf den sich die Messung bezieht. Dieser entspricht der Kontext-Angabe im OCL-Ausdruck.

In das Eingabefeld **acronym** wird ein noch nicht vorhandenes Kürzel eingetragen. Dieses Kürzel wird später genutzt, um die Messung in weiteren Messungen referenzieren und durchführen zu können.

Im Textfeld **enter ocl measure** wird der OCL-Ausdruck für die Messung angegeben. Dieser Ausdruck ist eine Operationsdefinition in OCL. Diese allgemeine Struktur wird in Listing 5.1 dargestellt.

```
context Classifier def: Operationsname(): Ergebnistyp = Ausdruck
```

Listing 5.1: Allgemeine OCL-Struktur für Messungen

Werden in der anzulegenden Messung OCL-Funktionen oder Messungen aus den Listen auf der rechten Seite verwendet, so ist zu beachten, dass diese nicht direkt im Feld **enter OCL measure** eingetippt, sondern per Drag-and-Drop hinzugefügt werden müssen. Hierdurch wird sichergestellt, dass entsprechende Messungen verwendbar sind.

Abschließend kann in dem Feld **informal definition** eine detaillierte Beschreibung zur Messung angegeben werden.

Nachdem der Anwender die Messung eingegeben und den Dialog mit [Save Changes] bestätigt hat, muss er die neu angelegte Messung dem fünften Qualitätsattribut per Drag-and-Drop zuordnen. Das fertig ausgefüllte Messmodell ist in Abbildung 5.62 abgebildet.

Das Messmodell ist somit fertiggestellt und die Messung des Modells kann beginnen. Um eine Messung durchzuführen, wählt der Anwender das UML/Z-Modell aus. Abbildung 5.63 zeigt den zugehörigen Auswahldialog.

Sobald der Anwender das Modell gewählt und einen Speicherort für die Messergebnisse gewählt hat **measurement results output**, kann er die Messung mit [do Measurement] durchführen.

5.2.2.6 Das Präsentationsmodell verwenden

Im Präsentationsmodell werden die Elemente des MQPs und die Messergebnisse angezeigt. Abbildung 5.64 zeigt das Präsentationsmodell für dieses Beispiel.

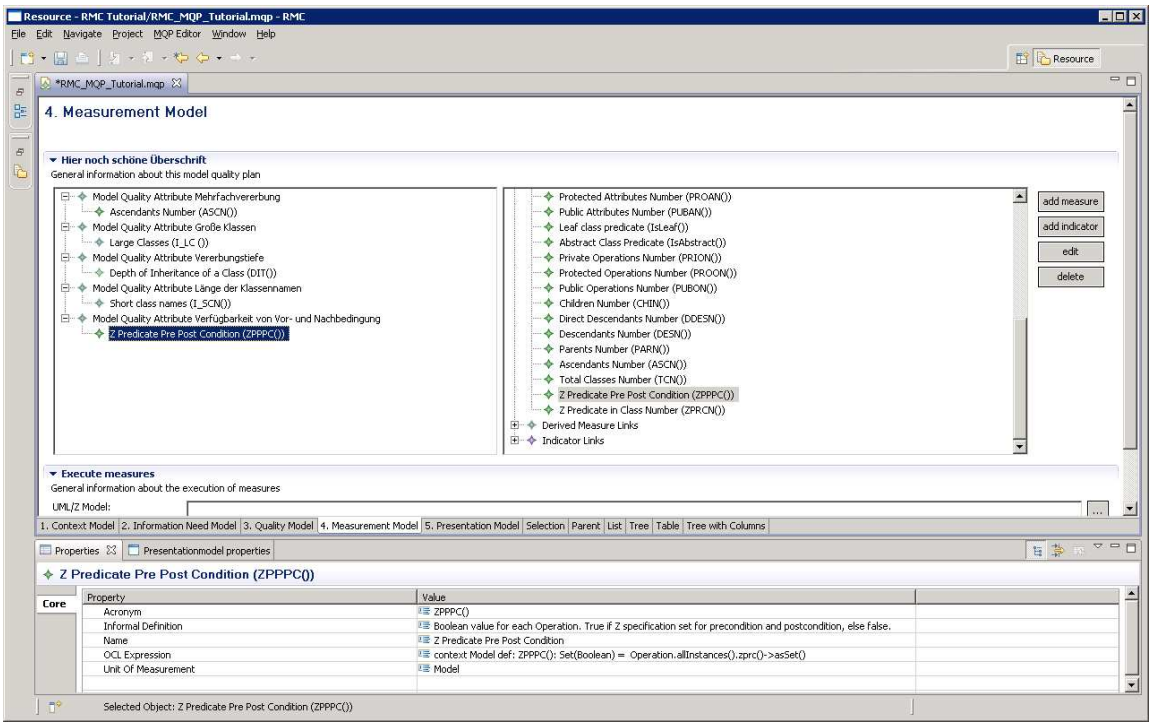

Resource - RMC Tutorial/RMC_MQP_Tutorial.mqp - RMC
File Edit Navigate Project MQP Editor Window Help
*RMC_MQP_Tutorial.mqp
4. Measurement Model
Hier noch schöne Überschrift
General information about this model quality plan
Model Quality Attribute Mehrfachvererbung
Ascendants Number (ASCN())
Model Quality Attribute Große Klassen
Large Classes (I_LC ())
Model Quality Attribute Vererbungstiefe
Depth of Inheritance of a Class (DIT())
Model Quality Attribute Länge der Klassennamen
Short class names (I_SCN())
Model Quality Attribute Verfügbarkeit von Vor- und Nachbedingung
Z Predicate Pre Post Condition (ZPPPC())
Protected Attributes Number (PROAN())
Public Attributes Number (PUBAN())
Leaf class predicate (IsLeaf())
Abstract Class Predicate (IsAbstract())
Private Operations Number (PRION())
Protected Operations Number (PROON())
Public Operations Number (PUBON())
Children Number (CHIN())
Direct Descendants Number (DDESN())
Descendants Number (DESN())
Parents Number (PARN())
Ascendants Number (ASCN())
Total Classes Number (TCN())
Z Predicate Pre Post Condition (ZPPPC())
Z Predicate in Class Number (ZPRCN())
Derived Measure Links
Indicator Links
add measure
add indicator
edit
delete
Execute measures
General information about the execution of measures
UML/Z Model:
1. Context Model 2. Information Need Model 3. Quality Model 4. Measurement Model 5. Presentation Model Selection Parent List Tree Table Tree with Columns
Properties
Presentationmodel properties
Z Predicate Pre Post Condition (ZPPPC())
Core
Property
Value
Acronym
ZPPPC()
Informal Definition
Boolean value for each Operation. True if Z specification set for precondition and postcondition, else false.
Name
Z Predicate Pre Post Condition
OCL Expression
context Model def: ZPPPC(): Set(Boolean) = Operation.allInstances().zprc()->asSet()
Unit Of Measurement
Model
Selected Object: Z Predicate Pre Post Condition (ZPPPC())

Abbildung 5.62: Messmodell

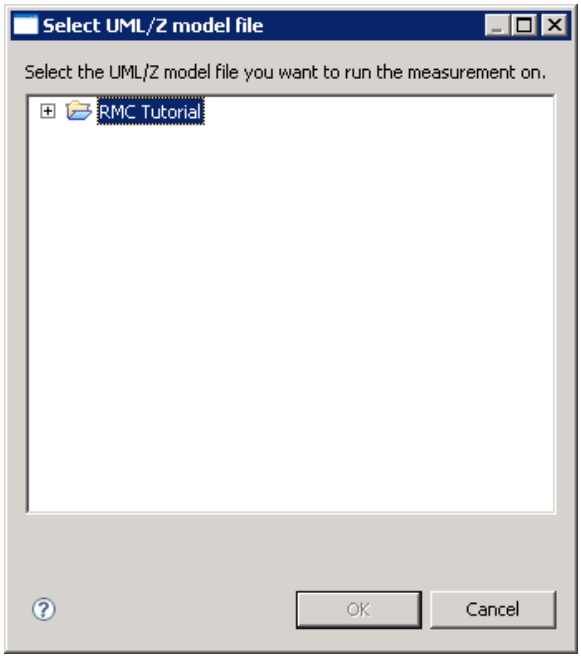


Abbildung 5.63: Auswahl des Modells

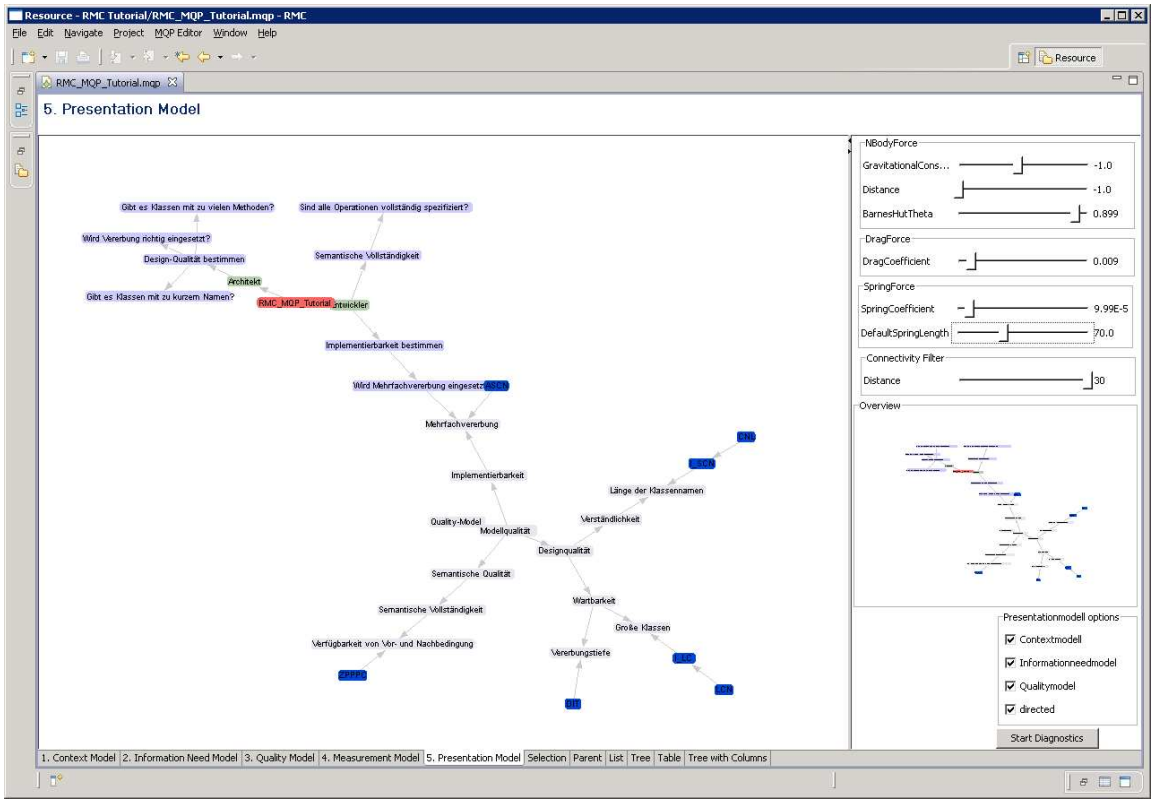


Abbildung 5.64: Tutorial: Präsentationsmodell

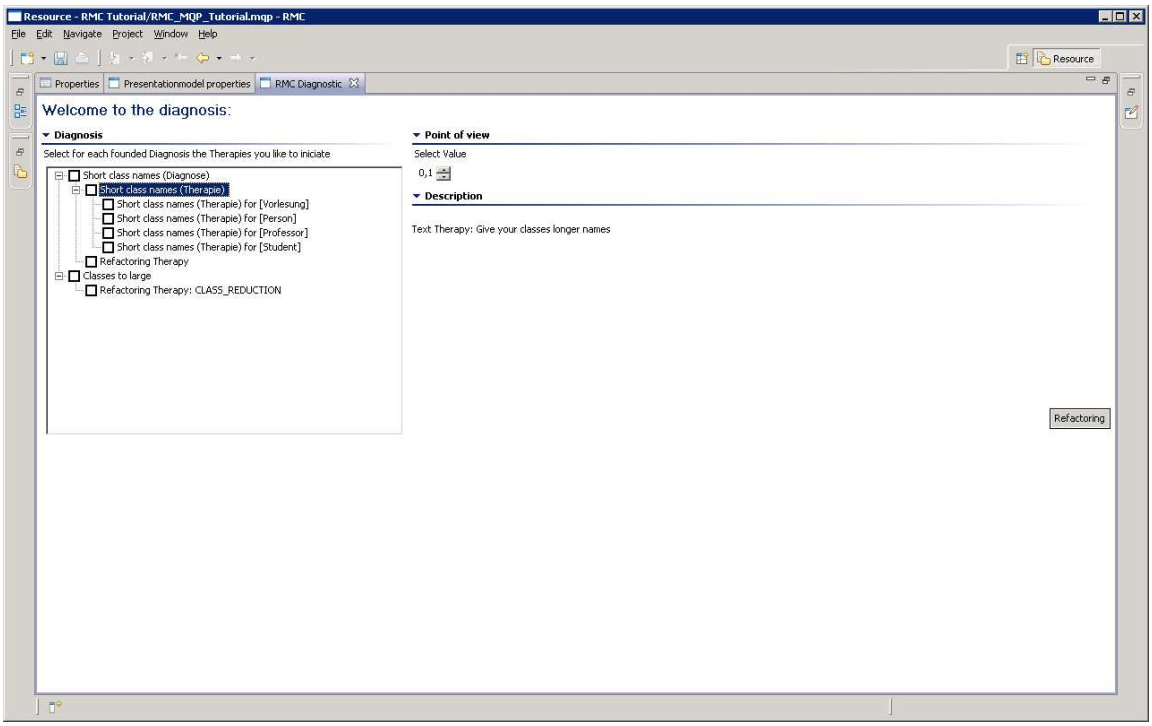


Abbildung 5.65: Benutzung der Diagnose

5.2.3 Diagnose

Bis jetzt hat der Benutzer ein neues Projekt angelegt und einen kompletten **Modellqualitätsplan** erstellt. Im folgenden Kapitel wird nun das vom Benutzer erstellte Diagramm auf Modelldefekte hin überprüft, und es werden gegebenenfalls Verbesserungen in Form von Therapien (Text- und/oder Refactoringtherapien) dem Benutzer angeboten. Abbildung 5.65 zeigt die Diagnose, nachdem der Button [Start diagnostic] auf der rechten Seite des Präsentationsmodells (siehe rechte Seite in Abbildung 5.64) betätigt wurde.

Im Abschnitt **Diagnosis** der Diagnose-GUI (Abbildung 5.65) werden dem Benutzer alle Modelldefekte angezeigt, deren normalisierte Messwerte gleich oder größer dem eingestellten Grenzwert sind[1]. Der Grenzwert, in der Abbildung mit (**Point of view**) bezeichnet, hat wie der Benutzer sehen kann den Wert 0,1. Da es sich bei diesem Wert um einen niedrigen Wert zwischen 0 und 1 (Grenzwerte der Normalisierung) handelt, bedeutet dies, dass auch nicht so schwere Modelldefekte in Form von Diagnosen und ggf. passenden Therapien dem Benutzer angezeigt werden. Setzt der Benutzer den Grenzwert höher an, so werden mit steigendem Grenzwert nur noch schwerwiegendere Modellmängel in Form von Diagnosen und Therapien angezeigt.

In Abbildung 5.65 kann der Benutzer bei einem Grenzwert von 0,1 nach Aufklappen der gesamten Baumstruktur im Abschnitt **Diagnosis** beispielsweise sehen, dass für die Klassen

[1] Vorraussetzung für die Anzeige von Modelldefekten ist, dass der Modelldefekt der Diagnosekomponente bekannt ist

Vorlesung, **Person**, **Professor** und **Student** Therapien angeboten werden. Betrachtet der Benutzer Abbildung 5.65 etwas genauer, so ist zu erkennen, dass für die Diagnose **Short class names** vier Therapien (jeweils für jede Klasse eine) angeboten werden. Bei diesen Therapien handelt es sich um Texttherapien, für die bei Anklicken auf den Namen z.B. auf den Ausdruck **Short class names (Therapie) for [Vorlesung]** im **Description**-Abschnitt eine textuelle Beschreibung einer Verbesserung gegeben wird. Klickt der Benutzer z.B. auf den String **Short class names(Therapie)** wie in Abbildung 5.65 zu sehen ist, werden im **Description**-Abschnitt dem Benutzer nähere Informationen zu den Therapien gegeben.

Für die gerade beschriebene Diagnose (**Short class names(Therapie)**) ist auf gleicher Ebene auch zu sehen, dass es hier keine Refactoringtherapie gibt. Würde es eine Refactoringtherapie geben, so wäre hinter dem String **Refactoring Therapy** in Großbuchstaben wie für die Diagnose **Classes to large** ein Refactoring wie beispielsweise **CLASS_REDUCTION** angezeigt. Wenn der Benutzer die Checkbox[2] vor der Refactoringtherapie auswählen würde, so könnte er durch Betätigung des Buttons [Refactoring] das Refactoring starten. Ab jetzt würde der Benutzer durch einen Wizard mit genaueren Informationen zur Durchführung des Refactorings versorgt und könnte dieses ausführen.

5.2.4 Refactoring

Auf den folgenden Seiten wird die Durchführung eines Refactorings gezeigt. Das Ziel ist die Klasse Professor um einige Funktionen zu erleichtern. Dazu wird die Klasse Professor selektiert und ⟨`Refactoring`⟩ ausgewählt. Nach der Auswahl von ⟨`Refactoring`⟩ werden die in der Konfigurationsdatei definierten Refactorings angezeigt. Alle nicht möglichen Refactorings sind dabei ausgegraut, so dass in diesem Fall nur das gewünschte **SplitClass-Refactoring** zur Auswahl steht (siehe Abbildung 5.66).

Nach der Auswahl von SplitClass öffnet sich der Refactoring-Wizard. Dort sind einige Informationen des durchzuführenden Refactorings zu finden. So steht in der Titelzeile der Startseite des Wizards das ausgewählte Refactoring: **SplitClass**. In der Mitte der Startseite ist die Beschreibung des Refactorings angegeben, in der die Eigenschaften des Refactorings nachgelesen werden können. Um das Refactoring auszuführen, ist der [Next]-Button anzuwählen, der den Dialog weiterführt (siehe Abbildung 5.67).

Auf der zweiten Seite steht in der Titelzeile neben dem Refactoring auch der Typ, in diesem Fall eine **selection decision**. In diesem Beispiel ist zwar nur die Klasse Professor auswählbar, sollten aber mehrere Klassen in der Tabelle aufgelistet sein, darf nur eine Klasse für die weiteren Schritte angewählt sein (siehe Abbildung 5.68).

Die Klasse Professor ist ausgewählt, und der Dialog kann fortgeführt werden (siehe Abbildung 5.69).

Auf der nächsten Dialogseite sind alle Methoden der Klasse Professor aufgelistet, und dem Nutzer obliegt es hier, die Methoden auszuwählen, die aus der Klasse Professor in eine neue Klasse verschoben werden sollen. Hier ist eine Besonderheit zu beachten: Wenn eine Methode verschoben wird, werden auch die Attribute dieser Methode mit verschoben. In

[2] Checkboxen haben nur für Refactoringtherapien eine Bedeutung

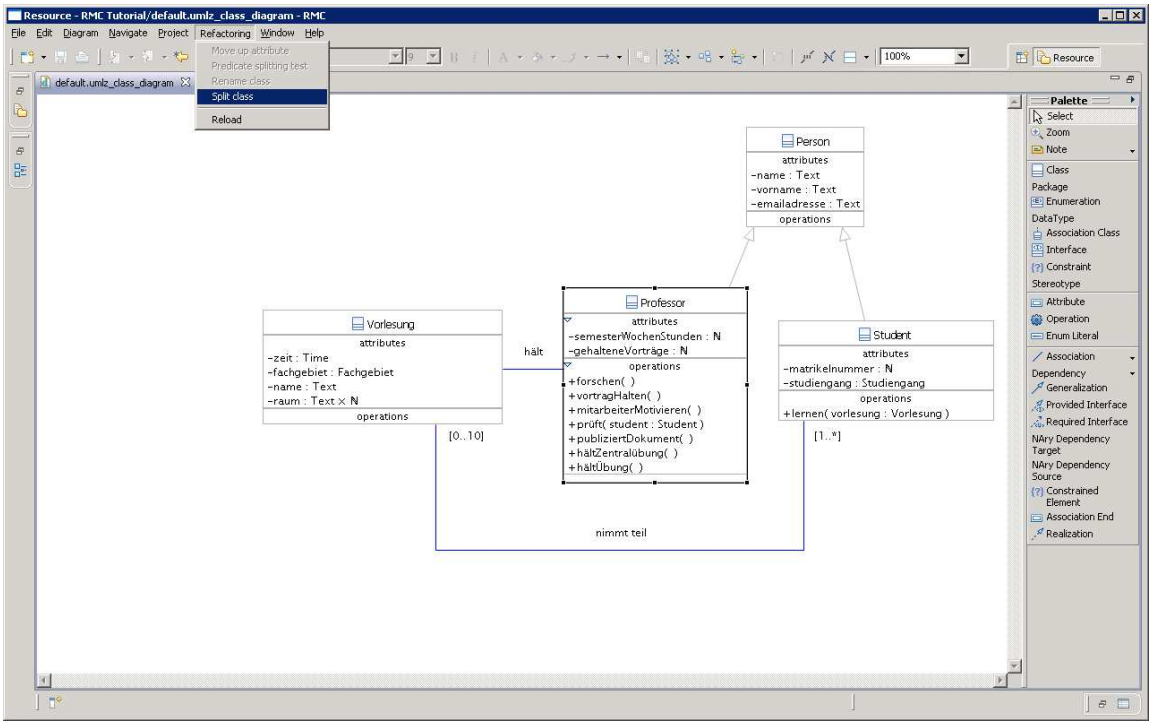


Abbildung 5.66: Auswahl des Refactorings SplitClass

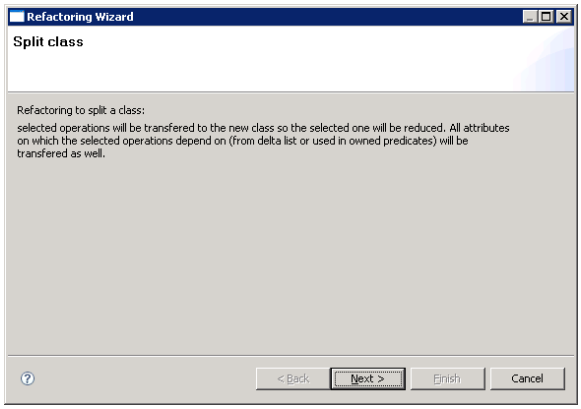

Refactoring Wizard
Split class
Refactoring to split a class:
selected operations will be transfered to the new class so the selected one will be reduced. All attributes on which the selected operations depend on (from delta list or used in owned predicates) will be transfered as well.
< Back
Next >
Finish
Cancel

Abbildung 5.67: Startseite des Wizards

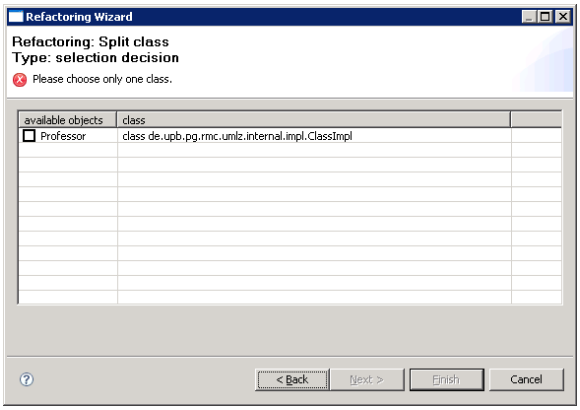


Abbildung 5.68: Seite mit dem Refactoring

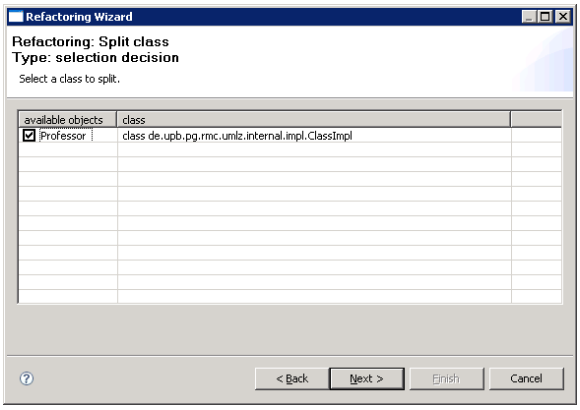


Abbildung 5.69: Auswahl der Klasse

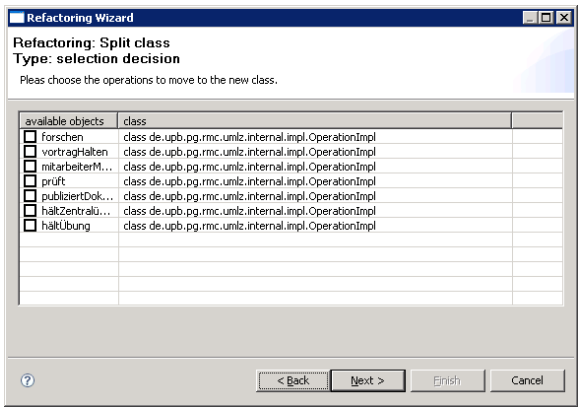


Abbildung 5.70: Methodenauswahl

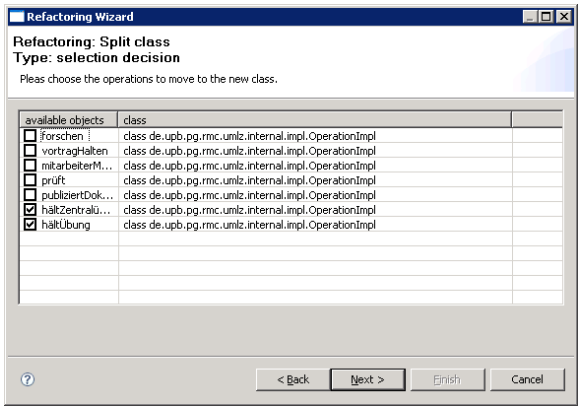


Abbildung 5.71: Selektion auszugliedernder Methoden

der Klasse darf kein Attribut verbleiben, welches sich in der Delta-Liste einer noch nicht selektierten Methode befindet. Erst wenn dies gewährleistet ist, kann der nächste Schritt ausgeführt werden (siehe Abbildung 5.70).

In diesem Beispiel werden die beiden Methoden `hältÜbung` und `hältZentralübung` für das weitere Vorgehen ausgewählt, der nächste Schritt des Dialogs kann nun angewählt werden (siehe Abbildung 5.71).

Die im letzten Schritt ausgewählten Funktionen werden in eine neue Klasse ausgelagert, die in diesem Dialogschritt angelegt wird. Aus diesem Grund ist der Typ der Entscheidung diesmal eine **naming decision** und keine **selection decision** mehr. Wichtig ist, einen sinnvollen Namen für die neue Klasse auszuwählen (siehe Abbildung 5.72).

Die nächsten beide Schritte sind für das Festlegen des Anfangs und des Endes der neuen Assoziation, die die neu angelegte Klasse mit der Klasse Professor verbindet, notwendig. Daher gibt man den Namen der Quelle und des Ziels für die Assoziation an (siehe Abbildung 5.73 und Abbildung 5.74).

Die letzte Seite des Dialogs zeigt den Fortschritt der Durchführung des Refactorings an, dazu dient der Fortschrittsbalken. Die Änderungen sind komplett durchgeführt worden,

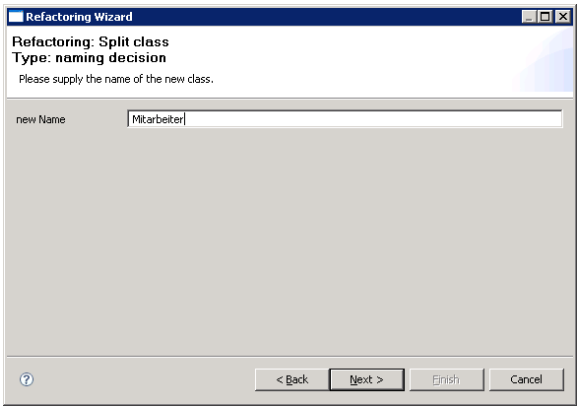


Abbildung 5.72: Benennung neuer Klasse

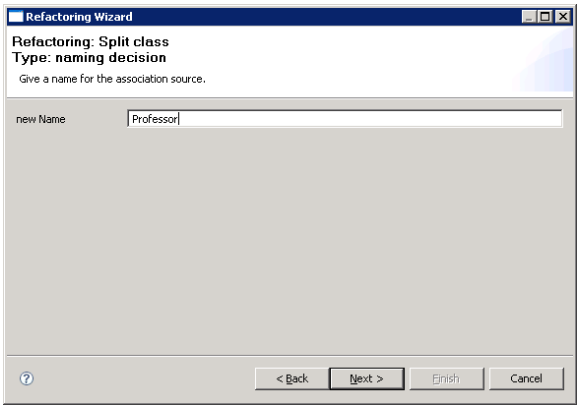


Abbildung 5.73: Beginn der Assoziation angeben

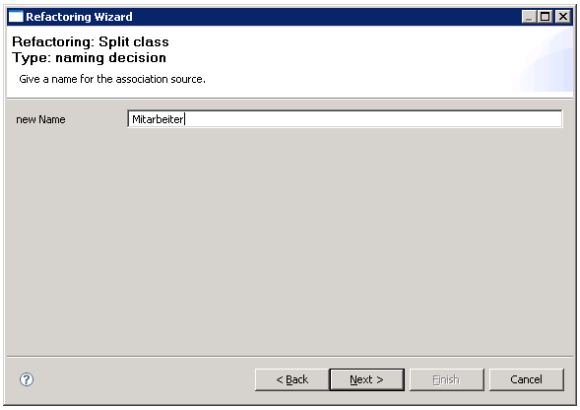


Abbildung 5.74: Ende der Assoziation angeben

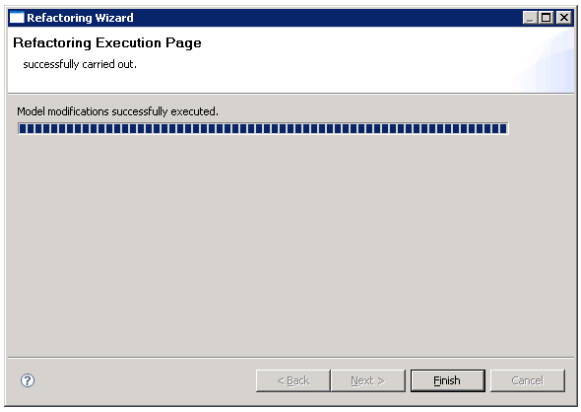


Abbildung 5.75: Refactoring durchgeführt

wenn der Fortschrittsbalken vollständig ist. Danach kann über den [Finish]-Button der Dialog geschlossen werden (siehe Abbildung 5.75).

Teil III

Entwicklerorientierte Sicht

6 Architekturdokumentation

> „Komm mal rüber, ich zeig dir das schweizer Taschenmesser der UML-Modellierungstools.“
>
> *Christian*

6.1 Komponentenübersicht

Dieses Kapitel beschreibt die Systemarchitektur von RMC, einem Tool, das eine Erstellung und Bearbeitung von Klassen- und Zustandsdiagrammen eines UML/Z-Modells sowie das Messen der Qualität erzeugter Modelle und die Verbesserung dieser erlaubt. Die Qualitätsmessungen und deren Aussage können mit RMC in Modellqualitätsplänen (MQPs) definiert werden. Nach einer Messung ist der Anwender in der Lage, festgestellte Modellmängel analysieren zu lassen und unter Verwendung konfigurierbarer Refactorings, also verhaltenserhaltender Modifikationen des Modells, diese zu beheben.

Aufgrund von unterschiedlichen Aufgabenbereichen wurden beim Entwurf von RMC geeignete Funktionalitäten in Komponenten zusammengefasst. Somit definiert die Komponente **UML/Z-Model** die Strukturen und Abläufe bei der Erzeugung und Änderung einzelner Elemente im Modell und die Komponente **UML/Z-Editor** stellt dem Anwender eine Benutzerschnittstelle mit angemessenen Werkzeugen zur Verfügung, ein entsprechendes UML/Z-Modell zu bearbeiten. Dabei werden Z-Annotationen unter Zuhilfenahme der Komponente **CZTWrapper** syntaktisch analysiert und in geeigneter Form zur Verfügung gestellt.

MQPs können in der Komponente **ModelQualityPlan** bearbeitet und auf UML/Z-Modellen ausgeführt werden. Nach einer Messung können deren Ergebnisse an die **Diagnostics**-Komponente geleitet werden, um diese zu analysieren und mögliche Verbesserungsvorschläge anzubringen. Dabei können Verbesserungsvorschläge auch Refactorings sein, die mit der Komponente **Refactoring** nicht nur ausgeführt, sondern auch definiert werden können.

Eine zentrale, für den Endanwender jedoch kaum erkennbare Rolle, spielt in RMC die Komponente **OCLWrapper**. Damit lassen sich OCL-Anfragen auf UML/Z-Modellelementen ausführen, so dass Definitionen nicht nur von Messungen, sondern auch von Refactorings über diese Komponente ermöglicht werden.

Der Abbildung 6.1 ist eine Übersicht zum grundlegenden Aufbau von RMC zu entnehmen. Das gewählte Design gewährleistet einen flexiblen Gebrauch und Austausch einzelner Komponenten.

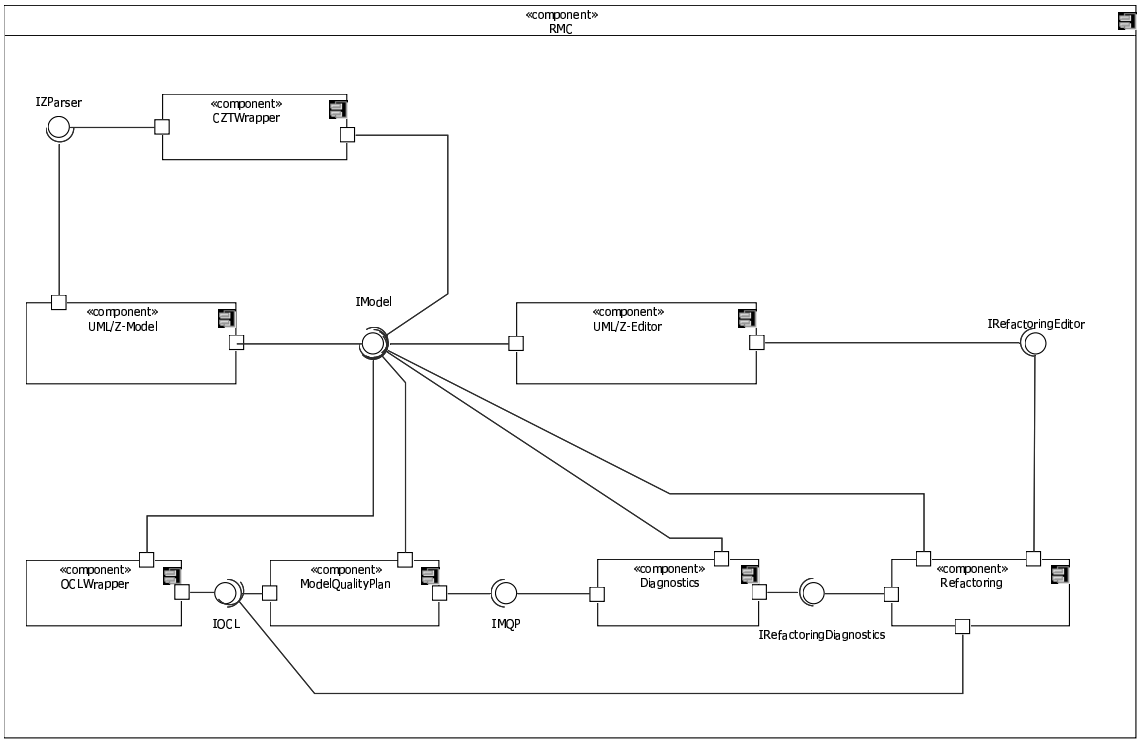


Abbildung 6.1: Komponentendiagramm: Architekturübersicht

Innerhalb von RMC verwendete Schnittstellen sind in Kapitel 6.2 detailliert dokumentiert. Nachstehend sind Zusammenfassungen der definierten Schnittstellen sowie deren Aufgaben gemäß Abbildung 6.1 angegeben. Schnittstellen werden jeweils von der Komponente angeboten, welche Klassen enthält, die die Schnittstellen implementieren.

IModel: Diese Schnittstelle stellt anderen Komponente Funktionalitäten zur Verfügung, auf Modellelemente zuzugreifen und so Zusammenhänge zwischen einzelnen Modellbestandteilen zu erfahren. Modifikationen des Modells können über eine Reihe definierter Operationen vorgenommen werden (siehe Unterkapitel 6.2.1).

IRefactoringEditor: Über diese Schnittstelle werden Refactorings zu einer Menge von Modellelementen angefordert. (siehe Unterkapitel 6.2.2).

IRefactoringDiagnostics: Über diese Schnittstelle werden Refactorings zu einer Menge von Modellelementen und einer Menge vorgegebener Effekte, die ein Refactoring herleiten sollte, angefordert. (siehe Unterkapitel 6.2.3).

IOCL: Diese Komponente überprüft gegebene OCL-Ausdrücke auf syntaktische Korrektheit. So erzeugte OCL-Anfragen können auf ein UML/Z-Modell angewandt werden, um Informationen zu bestimmten Modellelementen ausfindig zu machen (siehe Unterkapitel 6.2.4).

IMQP: Hierüber meldet sich die **Diagnostics**-Komponente an der **ModelQualityPlan**-Komponente an, um Indikatoren durchgeführter Messungen zu erhalten. Daraufhin bietet **Diagnostics** entsprechende Verbesserungen in einer eigenen GUI an (siehe Unterkapitel 6.2.5).

IZParser: Das UML/Z-Modell nutzt diese Schnittstelle, um Syntaxfehler in Z-Ausdrücken zu finden. Platzhalter in fehlerfreien Ausdrücken können dabei durch Elemente aus dem Modell ausgetauscht werden (siehe Unterkapitel 6.2.6).

In Unterkapitel 6.3 werden die Komponenten **ModelQualityPlan** (MQP), **Refactoring**, **UML/Z-Model**, **UML/Z-Editor** sowie die Komponenten **CZTWrapper** und **OCLWrapper** vorgestellt. Bestandteil dieses Kapitels sind Klassendiagramme, die Strukturelemente und Zusammenhänge zwischen diesen visualisieren, sowie Sequenzdiagramme, die die Kommunikation innerhalb der Komponenten beschreiben. Im Detail erfüllen die Komponenten jeweils folgende Aufgaben:

UML/Z-Model: Die Komponente **UML/Z-Model** beinhaltet das Metamodell der Sprache UML/Z. Des Weiteren bietet die Komponente Funktionalitäten zum Speichern eines UML/Z-Modells und Einlesen eines gespeicherten UML/Z-Modells an.

UML/Z-Editor: Der UML/Z-Editor ist das zentrale Element, um Änderungen an den vorliegenden Diagrammen vorzunehmen. Zudem ist auch ein Anlegen neuer Diagramme möglich.

ModelQualityPlan: In der Komponente **ModelQualityPlan** kann man eigene Maßstäbe für Qualität erstellen und auf Basis der aufgestellten Maßstäbe die Qualität von Klassen- und Zustandsdiagrammen messen.

Refactoring: Mit der Komponente **Refactoring** ist es möglich, das zu bearbeitende Diagramm mittels vordefinierter Regeln verhaltenserhaltend zu verändern.

Diagnostics: Die Komponente **Diagnostics** erhält Eingaben von der Komponente **ModelQualityPlan**. Auf Basis des Inputs erstellt diese eine Diagnose und schlägt geeignete Therapien vor.

CZTWrapper: Mit dem **CZTWrapper** werden Z-Ausdrücke analysiert und auf Konsistenz bezogen auf das dazugehörige UML/Z-Modell geprüft.

OCLWrapper: Der **OCLWrapper** bietet eine Möglichkeit an, um Anfragen zu Eigenschaften über Modellelemente bzw. Modellelementmengen an das Modell zu stellen. Eine weitere Aufgabe besteht darin, durch OCL-Anfragen das Modell zu ändern.

Einzelne Beschreibungen zu Komponenten von RMC berücksichtigen Abhängigkeiten externer Frameworks und Bibliotheken, die zum Zeitpunkt des Entwurfs feststanden. Dazu zählen die Komponenten **UML/Z-Model** (abhängig von Eclipse Modeling Framework [EMF]), **UML/Z-Editor** (abhängig von Graphical Modeling Framework [GMF]), **CZTWrapper** (abhängig von Community Z Tools [MU05]) und **OCLWrapper**.

6.2 Schnittstellen

6.2.1 IModel

Die Schnittstelle des UML/Z-Modells wird von allen Komponenten genutzt. Sie stellt Schnittstellen für sämtliche Klassen des UML/Z-Metamodells bereit. Die Schnittstelle bildet hierbei die Struktur (Vererbungshierarchie) des Modells ab. Diese entspricht den Strukturen der OMG UML2. Aufgrund ihres Umfangs sollen diese hier nicht erneut dokumentiert werden. Die Ergänzungen, die den Unterschied zwischen UML2 und UML/Z ausmachen, sind im Kapitel 6.3.1 beschrieben.

UML/Z-Editor: Der UML/Z-Editor, welcher durch Generierung mit dem Eclipse Graphical Modeling Framework (GMF) entstanden ist, baut direkt auf die Nutzung von EMF als Modellschicht auf. Dieser Zusammenhang der Komponenten ist in Abbildung 6.15 dargestellt. Die Kommunikation mit dem Modell geschieht sowohl direkt zwischen Editor und Modell als auch über die GMF Runtime. Weitere Informationen bietet die Modelldokumentation (Kapitel 6.3.1) und die GMF Hilfe [GMF].

ModelQualityPlan: Der MQP greift bei der Messung auf das Modell zu. Während die Messung an sich über den **OCLWrapper** abläuft, werden die von diesem zurückgegebenen Messergebnisse einzelnen Modellelementen zugeordnet.

Refactoring: In der **Refactoring**-Komponente werden Instanzen der Schnittstelle referenziert und im Ecore-Modell definierte Operationen mit Hilfe der **OCLWrapper**-Komponente (vergleiche 6.3.5 und 6.3.6) angestoßen.

CZTWrapper: Der **CZTWrapper** stellt Beziehungen zwischen Elementen analysierter Z-Ausdrücke und verfügbaren Elementen aus dem entsprechenden UML/Z-Modell unter Verwendung des **IModel**-Interfaces her. Auf diese Weise erzeugte Relationen zwischen Modellelementen und Variablen eines Z-Ausdrucks unterstützen die Komponente **UML/Z-Model** bei der Wahrung der Konsistenz.

OCLWrapper: Der **OCLWrapper** stellt einen Sonderfall beim Zugriff auf das Modell dar, da die OCL-Implementation für EMF nicht über das Interface des UML/Z-Modells auf das Modell zugreift, sondern nur auf im Ecore definierte Methoden zugreifen kann. Der Zugriff aus dem OCL-Plugin für EMF erfolgt dann über die Java Reflection API[1]. Die Methoden und Attribute, die für den Zugriff mit OCL auf das Modell zur Verfügung stehen, sind in der API beschrieben (siehe Kapitel 8).

6.2.2 IRefactoringEditor

In Abbildung 6.2 ist die Schnittstelle `IRefactoringEditor` dargestellt. Die **Refactoring**-Komponente bietet dem **UML/Z-Editor** die Funktionalität an, eine Liste möglicher Refactorings auf gegebenen Modellelementen zu erhalten. Es ist vorgesehen, dass der Editor auch ohne die **Refactoring**-Komponente funktioniert. Die Integration dieser Funktion in die GUI übernimmt der **UML/Z-Editor**.

[1] http://java.sun.com/javase/6/docs/api/

Der **UML/Z-Editor** kann über diese Schnittstelle eine Liste von `IRefactoringElement`-Objekten anfordern, die alle möglichen Refactorings beschreiben, die auf den übergebenen Modellelementen entsprechend der **Refactoring**-Konfigurationsdatei ausführbar sind. Der Editor kann ein Refactoring mittels dieser Objekte anstoßen, welches dann von der **Refactoring**-Komponente ausgeführt wird. Die restliche Nutzerinteraktion rund um das Refactoring erfolgt über die **Refactoring**-Komponente.

An einem `IRefactoringElement` angemeldete `IProgressListener` erhalten `IProgressEvents` mit Statusinformationen während der Durchführung eines Refactorings.

Ob ein Refactoring noch durchführbar ist, kann über `isApplicable` erfragt werden. Für die Gültigkeit dieser Angaben ist die **Refactoring**-Komponente verantwortlich und nicht der Editor.

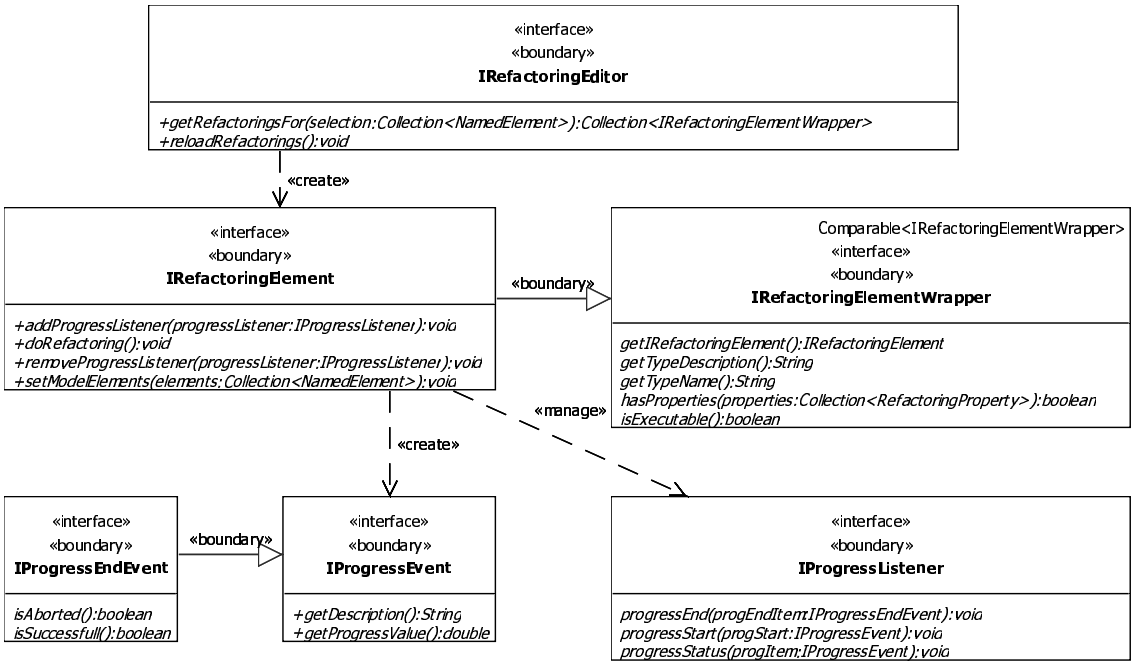


Abbildung 6.2: Klassendiagramm: IRefactoringEditor-Interface

6.2.3 IRefactoringDiagnostics

Über die **IRefactoringDiagnostics**-Schnittstelle werden Refactorings als Verbesserungsvorschläge von der **Diagnostics**-Komponente an die **Refactoring**-Komponente angefordert. Die **Diagnostics**-Komponente übergibt in `getRefactoringsFor` Modellelemente sowie Auswirkungen, die ein Refactoring haben sollte, um eine Verbesserung des Modells zu bewirken. Als Ergebnis bekommt die **Diagnostics**-Komponente eine Menge von `IRefactoringElement`-Objekten.

Die Interaktion während der Vorbereitung und der Durchführung des Refactorings übernimmt ausnahmslos die **Refactoring**-Komponente. Daher stößt die Diagnose diesen Prozess nur an und erhält nach Abschluss Informationen, ob das Refactoring korrekt durchgeführt wurde.

Die Struktur der Schnittstelle ist der Abbildung 6.3 zu entnehmen.

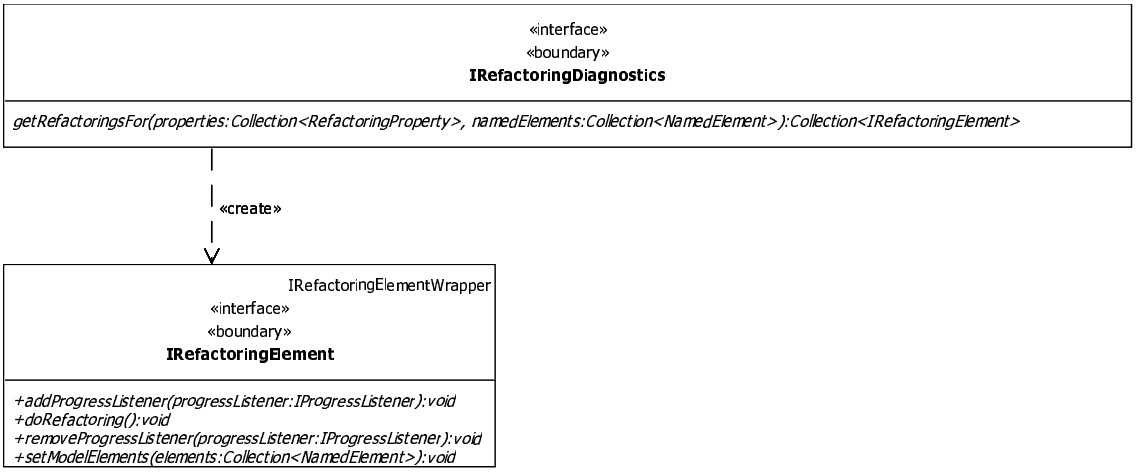


Abbildung 6.3: Klassendiagramm: IRefactoringDiagnostics-Interface

6.2.4 IOCL

Die Schnittstelle `IOCLParser` aus Abbildung 6.4 dient als Einstiegspunkt für andere Komponenten, um an eine OCL-Umgebung zu gelangen. Instanzen von `IOCLParseEnvironment` verwalten Variablen, die in OCL-Anfragen (`IOCLQuery`) verwendet werden können.

Mittels `execute` wird die eigentliche `IOCLQuery` durchgeführt. Falls diese einen Wert zurückliefert, der nicht kompatibel zum angeforderten Typ ist, wird eine `OCLTypeMismatchException` geworfen. Die Schnittstelle ist in Abbildung 6.4 dargestellt.

6.2.5 IMQP

Die Schnittstelle aus Abbildung 6.5 wird vom der **ModelQualityPlan**-Komponente genutzt, um der **Diagnostics**-Komponente die Ergebnisse einer Messung zu übergeben.

Die **MQP**-Komponente führt eine Messung durch und stellt die Ergebnisse über `IMQP` der **Diagnostics**-Komponente als Menge von `IMQPIndicator`-Objekten zur Verfügung. Die **Diagnostics**-Komponente erstellt auf Basis einer Eingabe Verbesserungsvorschläge und bietet diese in einer eigenen GUI an. Das Erstellen und die Durchführung der Verbesserungsvorschläge findet vollständig auf Seiten der Diagnose statt.

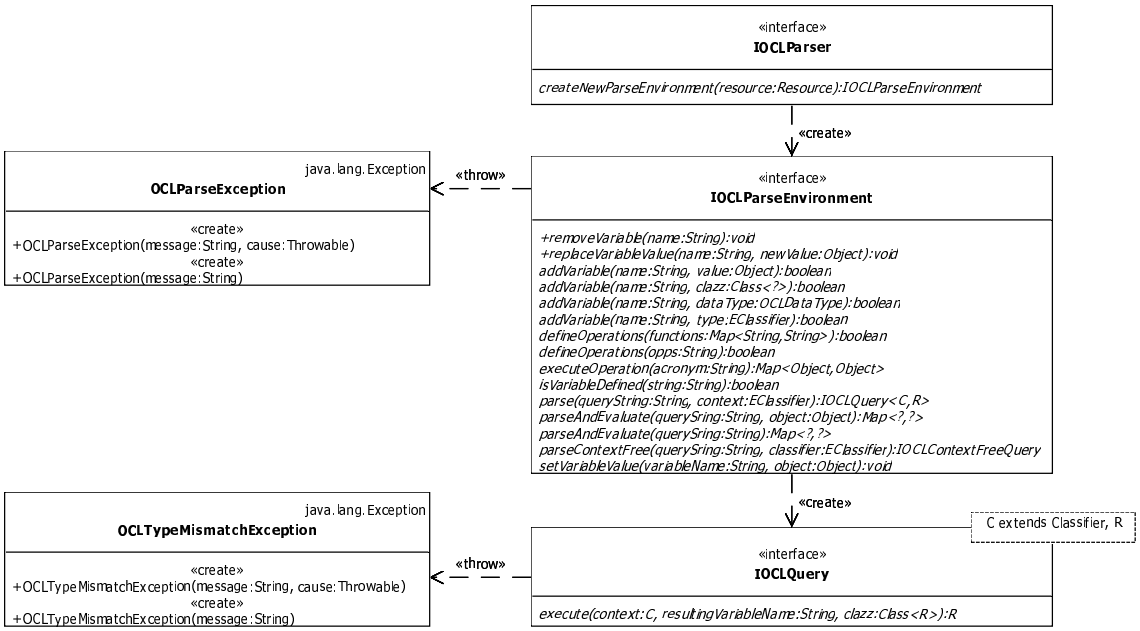


Abbildung 6.4: Klassendiagramm: IOCL-Interface

Die Skala eines `IMQPIndicators`, die den Rahmen angibt, in dem das Meßergebnis einzuordnen ist, wurde in Form der Schnittstelle `IMQPScale` ausgegliedert, damit sie unabhängig vom restlichen Interface angepasst werden kann. Die Methode `getTypeOfScale` liefert den Typ einer Skala. Nähere Informationen zu möglichen Skalen sind in Kapiteln 6.3.3 und 6.3.4 zu finden.

6.2.6 IZParser

IZParser beschreibt die Schnittstelle zwischen der **UML/Z-Model**-Komponente und dem **CZTWrapper**. Die Schnittstelle sieht vor, dass Zeichenfolgen unter Zuhilfenahme des `ICZTUtil` zu Z-Ausdrücken geparst werden. Damit wird die syntaktische Korrektheit der Angaben in Zeichenfolgen überprüft. Außerdem können auf diese Weise Teile des analysierten Ausdrucks, der anschließend als Abstract Syntax Trees (AST) vorliegt, unter Referenzierung von Modellelementen ersetzt werden, so dass bei Eigenschaftsänderungen der Modellelemente diese auch im Z-Ausdruck berücksichtigt werden.

Syntaktisch korrekte Z-Ausdrücke werden als Instanzen von `ITermWrapper` referenziert. Dabei stellen `IPredWrapper` Prädikate von Operationen einer UML/Z-Klasse dar. Sowohl **enable**- als auch **effect**-Prädikate können auf diese Weise angegeben werden. `IExprWrapper` beziehen sich auf Z-Ausdrücke, die Mengen und Beziehungen zwischen diesen veranschaulichen und werden verwendet, um beispielsweise neue Typen zu definieren ([WD96], [MU05]).

Über die Schnittstelle `ICZTUtil` werden sowohl `IPredWrapper` als auch `IExprWrapper` erzeugt.

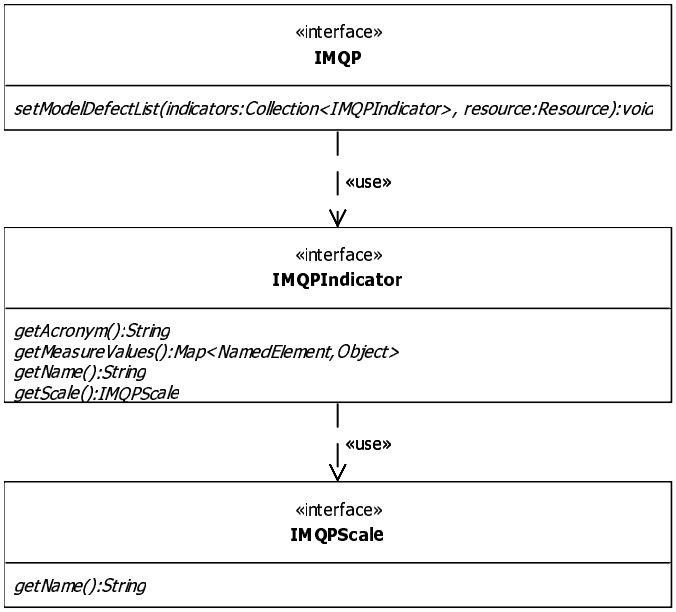


Abbildung 6.5: Klassendiagramm: IMQP-Interface

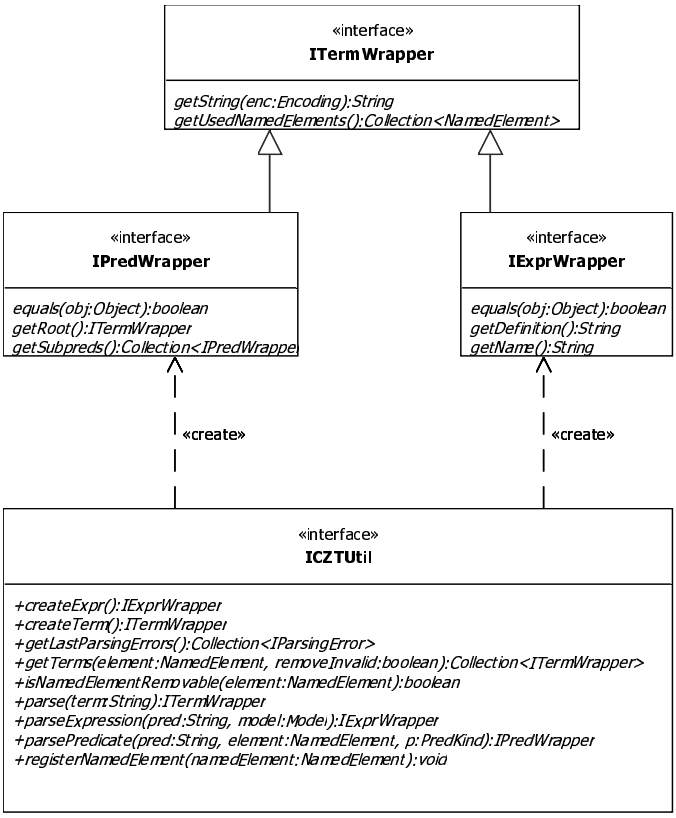


Abbildung 6.6: Klassendiagramm: IZParser-Interface

6.3 Komponenten

6.3.1 UML/Z-Model

Die Implementierung des UML/Z-Modells basiert auf der UML2-Implementierung von eclipse.org [UML], welche mit Hilfe des Eclipse Modelling Frameworks (EMF) [EMF] realisiert wurde. Wie bei allen EMF-Projekten besteht der generierte Java-Code aus vier Projekten:

- der Modell-Implementierung
- einem Hilfsprojekt (`edit`) zum Bearbeiten des Modells
- einem Baumeditor zum Bearbeiten und Erstellen von Modellen (`editor`)
- einem Projekt, mit welchem man Unit-Tests [HL03] auf dem generierten Code, bzw. eigenen Ergänzungen vornehmen kann.

Nicht der gesamte benötigte Code für ein funktionierendes Modell wird von EMF generiert. Insbesondere die Komplexität der UML2 erfordert etliche Anpassungen. Daher wurde das UML/Z-Modell auf Basis der bestehenden UML2-Implementierung erstellt. Für die Einbindung von Z waren weitere manuelle Anpassungen erforderlich.

Diese manuellen Anpassung sind im Kapitel 7 der Entwicklerdokumentation beschrieben.

Im Folgenden soll auf die UML/Z spezifischen architektonischen Besonderheiten der ersten drei der generierten Modellprojekte eingegangen werden. Die Beschreibung der von der Eclipse UML2 und EMF vorgegebenen Architektur findet sich in den entsprechenden Dokumentationen [UML] und soll hier nicht erläutert werden.

Die Modell-Implementierung (Paket `de.upb.pg.rmc.umlz`)

Die Modellimplementierung besteht aus insgesamt neun Paketen. Diese sind in Abbildung 6.7 mit ihren Abhängigkeiten als Paketdiagramm dargestellt. Die Pakete `umlz.izparser`, `umlz.types` und `umlz.cztutilprovider` sind hierbei spezifisch für das UML/Z-Modell. Die anderen Pakete stammen aus der Eclipse UML2 und wurden nur entsprechend ergänzt.

Das Paket `umlz` enthält neben den restlichen Unterpaketen Interfaces für sämtliche Klassen des Modells. Dies ist in erster Linie erforderlich, um Mehrfachvererbungen realisieren zu können. Im Paket `internal.impl` sind diese Interfaces dann implementiert. Während dieses Paket sämtliche automatisch generierte[2] Modellstrukturen wie **Collections** und **Getter**- und **Setter**-Methoden enthält, befinden sich alle im Ecore-Modell definierten Operationen im Paket `internal.operations`. Für die Methoden in diesem Paket generiert EMF nur Stubs, die dann manuell implementiert werden müssen. Hilfe zu weiteren EMF spezifischen Hilfspaketen bietet [EMF].

Im Paket `umlz.types` befinden sich die beiden primitiven Typen `ZExpression` und `ZPredicate`, die die Z-Ausdrücke aufnehmen, welche vom **CZTWrapper** ausgewertet werden.

[2] Auch in diesem Paket befinden sich einige manuelle Anpassungen.

Der Zugriff auf den **CZTWrapper** erfolgt mit Hilfe des Paketes `umlz.cztutilprovider`. Das Paket `umlz.izparser` enthält in Abschnitt 6.2.6 beschriebene Interfaces.

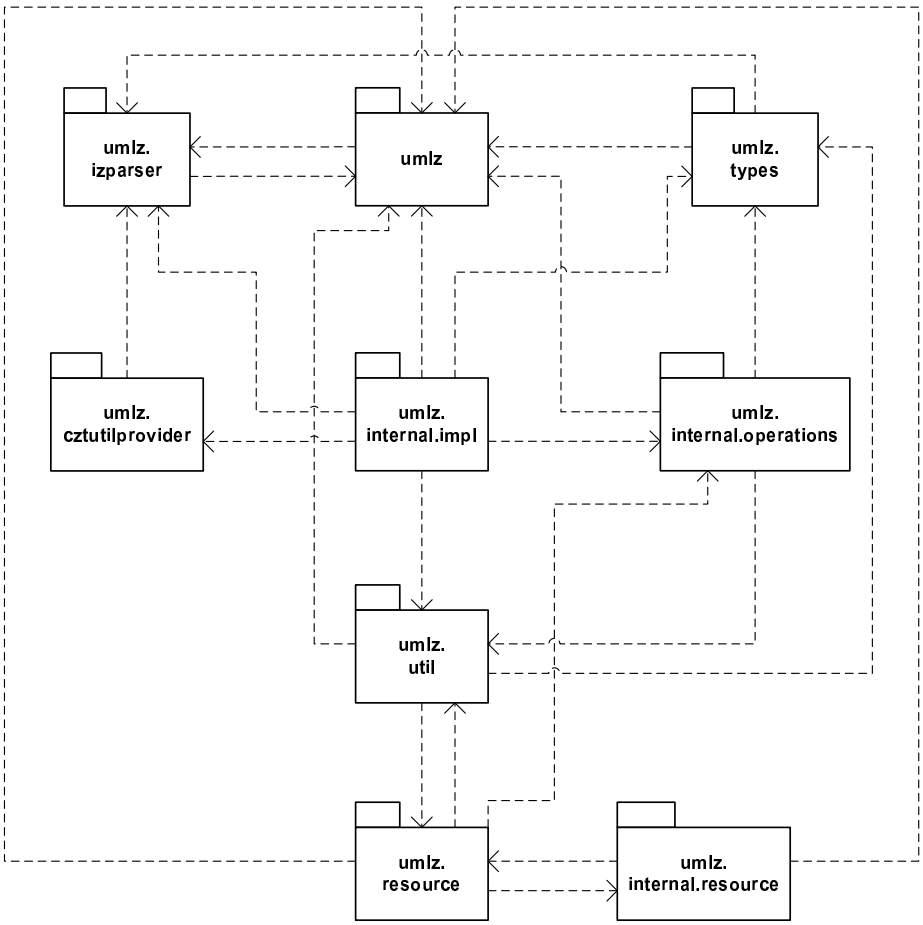


Abbildung 6.7: Paketdiagramm: UML/Z-Model

Das Hilfsprojekt zum Bearbeiten des Modells (Paket `de.upb.pg.rmc.umlz.edit`)

Das Hilfsprojekt zum Bearbeiten des Modells stellt im Paket `umlz.edit.providers` Hilfsmethoden wie Icon- und Label-Provider bereit, welche die Entwicklung eines Editors für das Modell vereinfachen. Abbildung 6.8 zeigt das entsprechende Paketdiagramm. Abhängigkeiten zu Paketen der Modellimplementierung sind grau dargestellt.

Der generierte Baumeditor (Paket `de.upb.pg.rmc.umlz.editor`)

Dieses Projekt stellt einen von EMF generierten Baumeditor für das Modell zur Verfügung. Ergänzt wurde es um das Paket `umlz.editor.z`, welches das Parsen der Z-Ausdrücke über den **CZTWrapper** anstößt. Abbildung 6.9 zeigt das entsprechende Paketdiagramm.

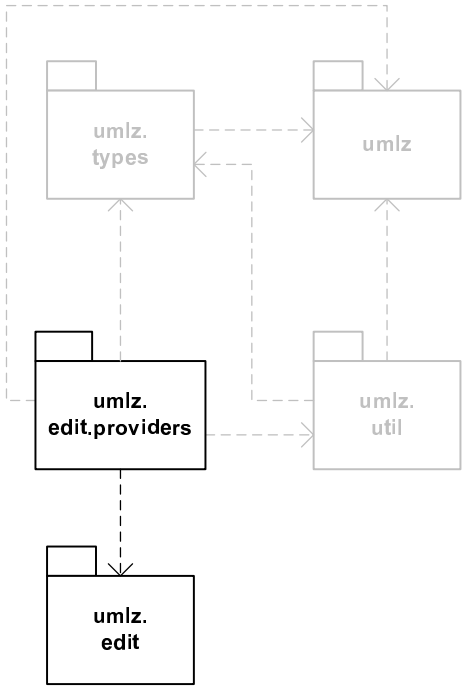


Abbildung 6.8: Paketdiagramm: UML/Z-Model-Edit

Abhängigkeiten zu Paketen der Modellimplementierung und dem Hilfsprojekt (`umlz.edit`) sind grau dargestellt.

6.3.1.1 Ergänzung der UML2 zu UML/Z

Im Folgenden sind Erweiterungen und Ergänzungen dokumentiert, die an der UML2 vorgenommen wurden. Das Augenmerk bei der Durchführung der Ergänzungen lag darauf, möglichst viele bereits bestehende Strukturen der UML2 weiterzuverwenden.

Operationen bzw. Operationsschemata

Enable und effect: Die Eigenschaften **enable** und **effect** einer Operation in Z entsprechen den Assoziationen **pre-** und **postcondition** in UML. In der UML ist es bereits vorgesehen, hier neben natürlichsprachlichen Bedingungen mit Hilfe der OCL auch formale Bedingungen zu definieren. Über die neu eingeführte Klasse `ZPredicateSpecification` kann hier jetzt ein Z-Prädikat angegeben werden (siehe Abbildung 6.10).

input, simple, output: Die Enumeration `ParameterDirectionKind` wurde um den Wert **simple** ergänzt. Für **input** und **output** aus Z werden die bestehenden Werte **in** und **out** aus der UML verwendet.

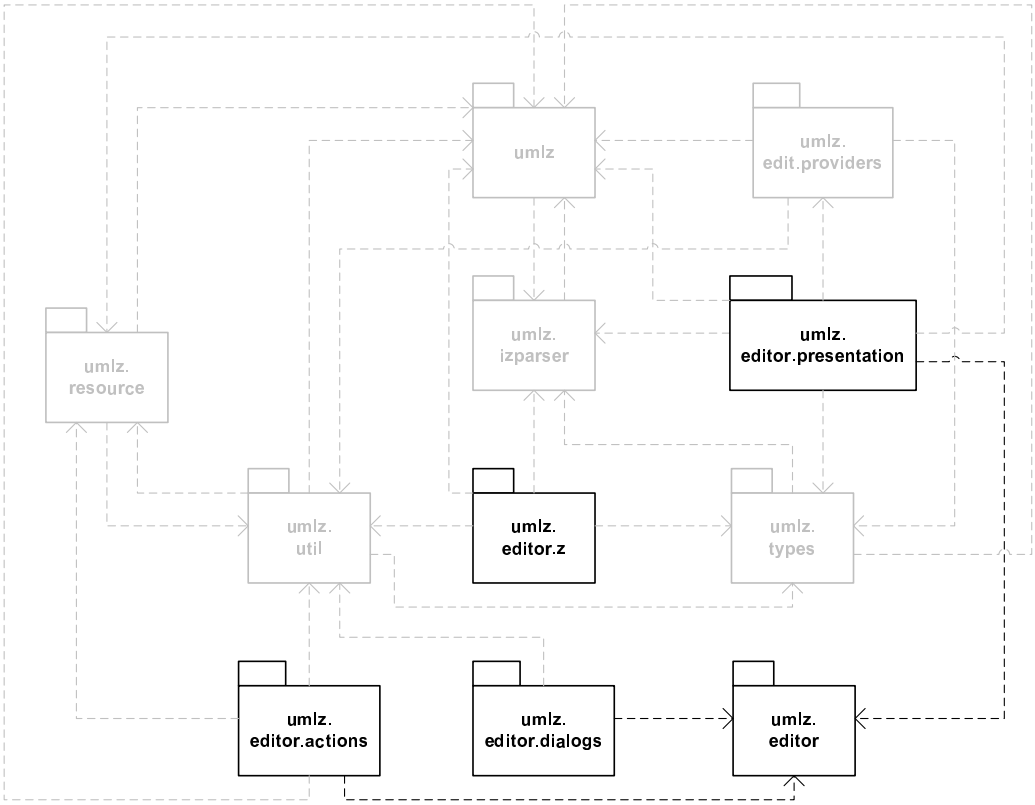


Abbildung 6.9: Paketdiagramm: UML/Z-Model-Editor

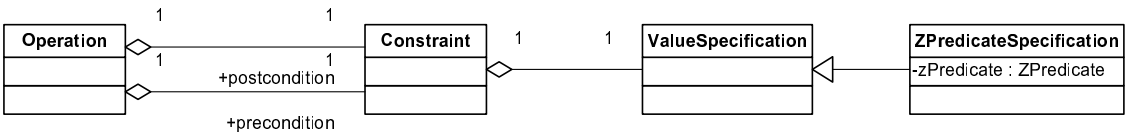


Abbildung 6.10: Klassendiagramm: Realisierung von enable- und effect-Bedingungen

Delta-Liste: Zur Spezifikation der Delta-Liste einer Operation wurde dieser eine zusätzliche Assoziation zu `Property` hinzugefügt, mit der die Delta-Attribute einer Klasse gemäß Z-Spezifikation [WD96] angegeben werden können (siehe Abbildung 6.11).

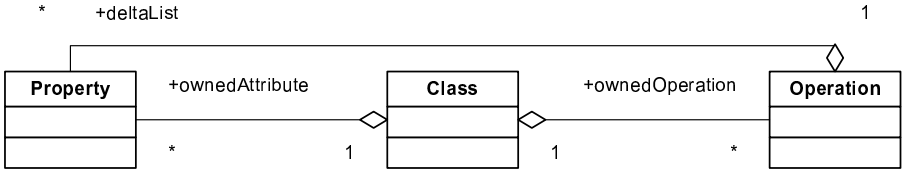


Abbildung 6.11: Klassendiagramm: Realisierung der Delta-Liste in Operationen

Z-Konstruktor

Für den Z-Konstruktor (**Init**) wurde der Klasse `Class` eine neue Assoziation **Init** hinzugefügt, die auf einen `Constraint` verweist, welches wiederum eine `ZPredicateSpecification` besitzt. In dieser kann wie bei der **effect** und **enable** ein Z-Prädikat für den Z-Konstruktor angegeben werden (siehe Abbildung 6.12).

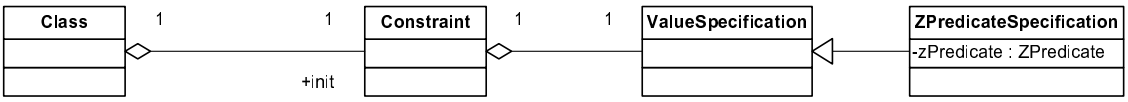


Abbildung 6.12: Klassendiagramm: Z-Konstruktor

Globale Z-Defintionen und Z-Typen

Z-Typen können mit Hilfe der Klasse `TypeDeclaration` angelegt werden. Ähnlich wie bei **enable** und **effect**, bzw. dem Z-Konstruktor wird der `TypeDeclaration` eine `ZExpressionSpecification` zugeordnet, die in einer `ZExpression` den Z-Ausdruck enthält.

Z-Typen die in einem `Package` angelegt wurden, sind jeweils nur innerhalb dieses `Package`s definiert.

Entsprechend der `TypeDeclaration` können mit der `VariableDeclaration` Variablen und mit der `FunctionDeclaration` Funktionen mit Z-Ausdrücken definiert werden (siehe Abbildung 6.13).

Zustandsdiagramme

Da die **enable**-Eigenschaft einer zu einer Transition gehörigen Operation die Bedingung für das Feuern der Transition sein soll, wurden die aus der UML2 bekannten Attribute **trigger** und **guard** gelöscht.

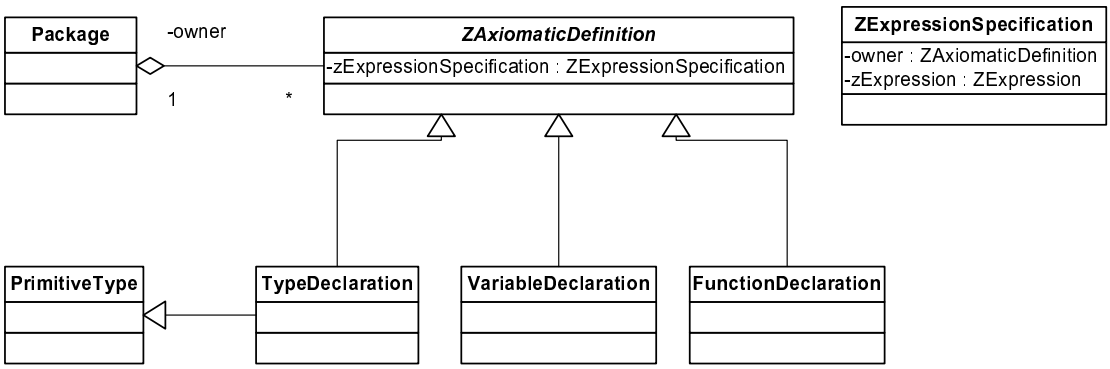


Abbildung 6.13: Klassendiagramm: Globale Z-Defintionen

6.3.1.2 Anbindung des Z-Parsers an das Modell

Abbildung 6.14 zeigt die Anbindung des Modells an den Z-Parser. Über den `INameWrapper` greift der Z-Parser auf benannte Modellelemente zu. Über die Interfaces `IExprWrapper` und `IPredWrapper` wird die Logik der Z-Ausdrücke verwaltet.
Eine genauere Beschreibung der Parser-Anbindung befindet sich zum einen im Kapitel zu **IZParser** sowie im Kapitel zum **CZT-Wrapper**.

6.3.2 UML/Z-Editor

Die grafischen Editoren basieren auf dem Graphical Modelling Framework [GMF] . Dieses stellt eine flexible Möglichkeit dar, um grafische Anwendungen auf EMF basierten Modellen zu entwickeln. Genutzt werden folgende Frameworks:

- Eclipse Modelling Framework (EMF) als Domänenmodell [EMF]
- Graphical Editing Framework (GEF) für die grafische Darstellung [GMF]

Die GMF Anwendung kombiniert das EMF-Modell mit einem Notationsmodell für die Darstellung. Der generierte Editor nutzt sowohl EMF und GEF direkt als auch die GMF Runtime. Tiefergehende Informationen über den Aufbau von GMF bietet die GMF-Dokumentation [GMF].

Die Realisierung der Editoren basiert in den Grundlagen auf der Implementierung der UML2 Tools [UML]. Abbildung 6.15 stellt den Zusammenhang eines GMF basierten Editors dar. Die Komponente **Graphical Editor** ist der generierte Editor. Für die beiden unterstützten Diagrammtypen gibt es je einen eigenen Editor. Die beiden folgenden Kapitel gehen auf dessen grundlegenden Aufbau und ihre Pakete ein. Einen Überblick über die benutzten Modelle und die Generierung des Quelltextes bietet Kapitel 7.2.2.1 der Entwicklerdokumentation.

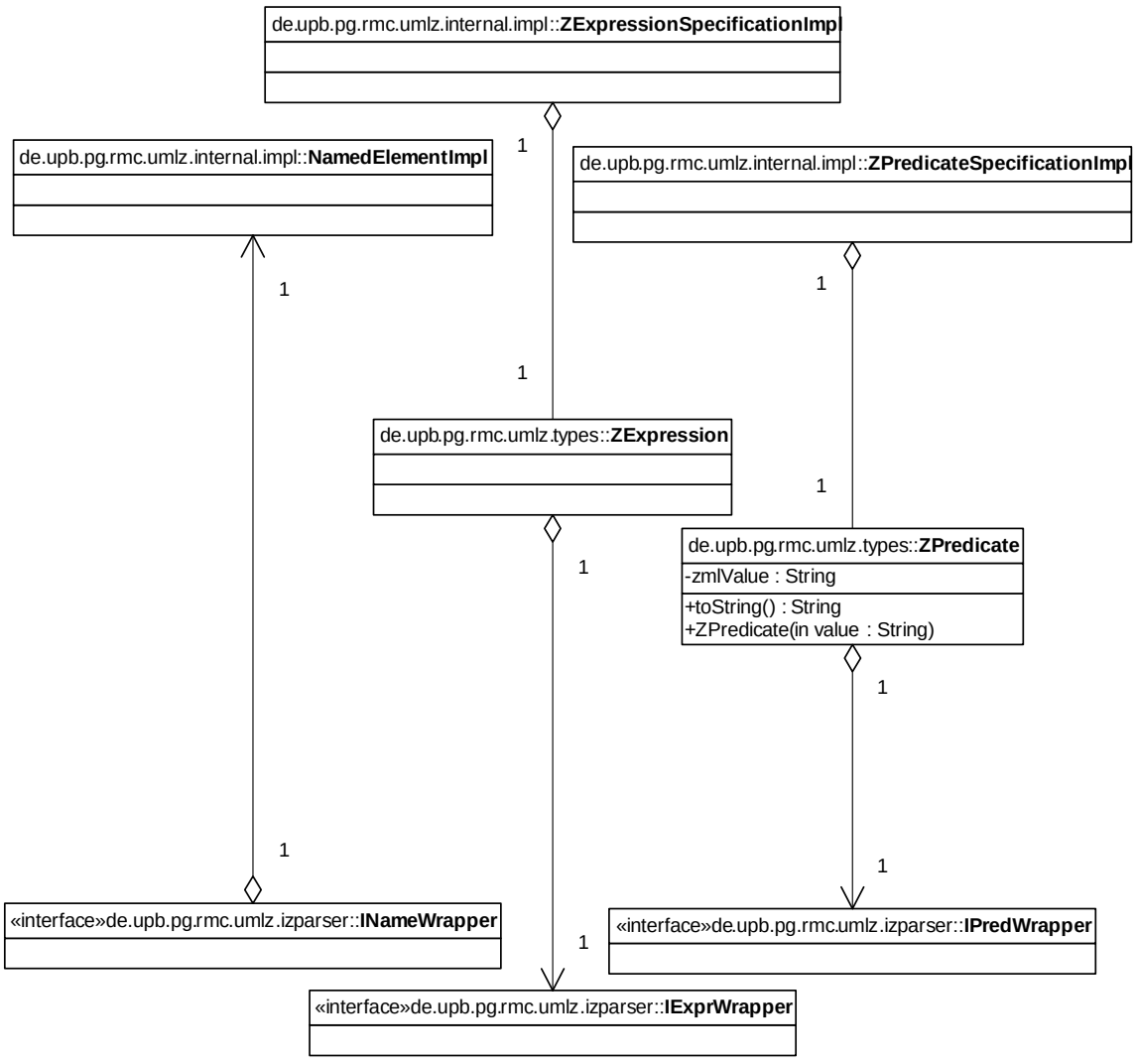


Abbildung 6.14: Klassendiagramm: Anbindung des Z-Parsers an das Modell

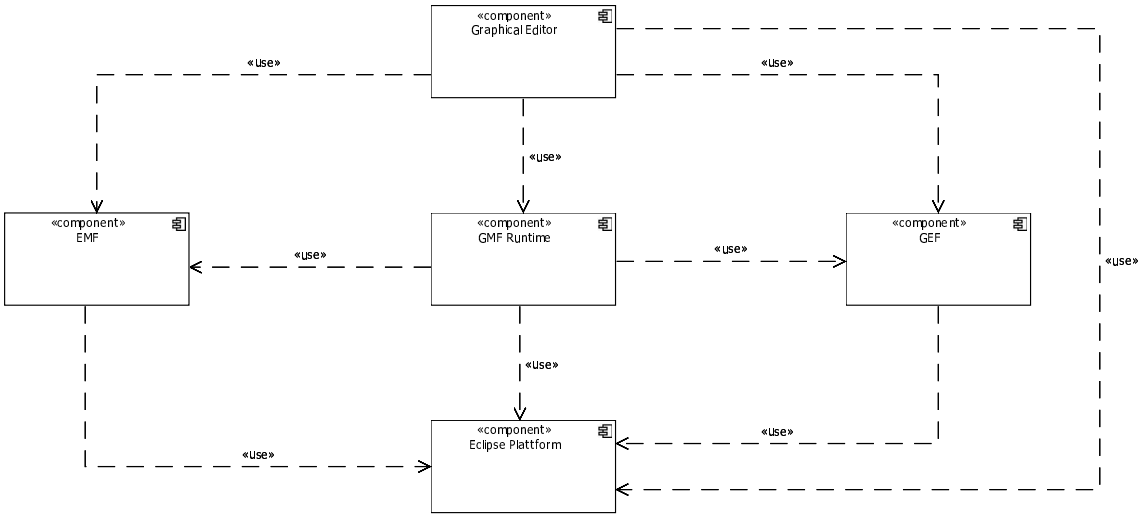


Abbildung 6.15: Übersicht über die Pakete eines GMF-basierten Editors [GMF]

6.3.2.1 Klassendiagrammeditor

Mittels Generierung durch GMF entsteht der Editor. Der generierte Quelltext ist noch in einigen Bereichen an die eigenen Bedürfnisse anzupassen. Die Struktur wird jedoch durch diese Änderungen nicht angetastet. Abbildung 6.16 zeigt die Pakete des Klassendiagrammeditors. Die Modifikationen und Erweiterungen des Quellcodes entsprechen zu weiten Teilen den UML2 Tools.

Auf die einzelnen Pakete und Abhängigkeiten soll nicht weiter eingegangen werden, da diese durch GMF vorgegeben sind. Informationen finden sich auf der GMF Webseite [GMF].

6.3.2.2 Zustandsdiagrammeditor

Auch der Zustandsdiagrammeditor ist durch Generierung mittels GMF entstanden. Auch hier sind nur Anpassung an die eigenen Anforderungen nötig. Abbildung 6.17 zeigt die Pakete des Editors. Die Modifikationen und Erweiterungen des Quellcodes entsprechen zu weiten Teilen den UML2 Tools.

Auf die einzelnen Pakete und Abhängigkeiten soll nicht weiter eingegangen werden, da diese durch GMF vorgegeben sind. Informationen finden sich auf der GMF Webseite [GMF].

6.3.3 ModelQualityPlan

Die Komponente **ModelQualityPlan** ist dafür zuständig, dem Anwender die Möglichkeit zu bieten, einen Modellqualitätsplan Schritt für Schritt anzulegen. Außerdem können damit Messungen auf einem UML/Z-Modell durchgeführt, resultierende Messergebnisse präsentiert und Messergebnisse unter Umständen der **Diagnostics**-Komponente übergeben werden.

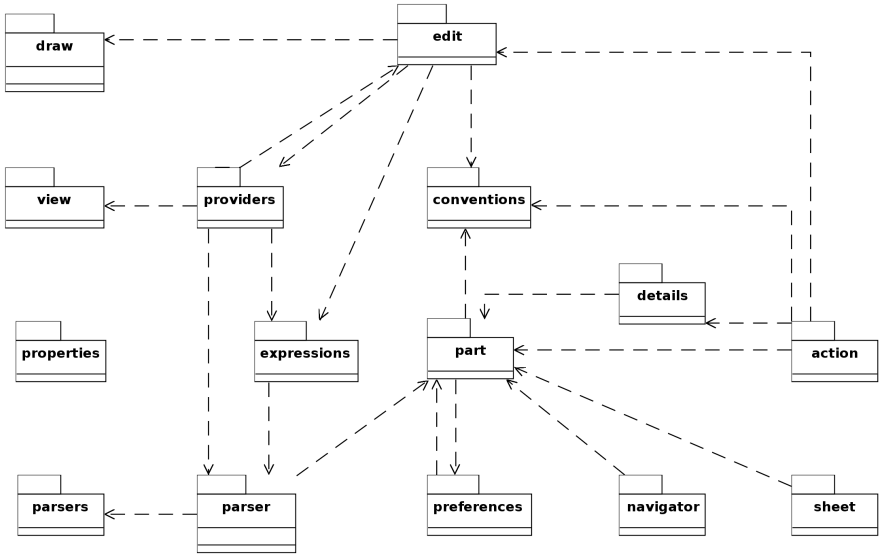


Abbildung 6.16: Pakete des Klassendiagrammeditors

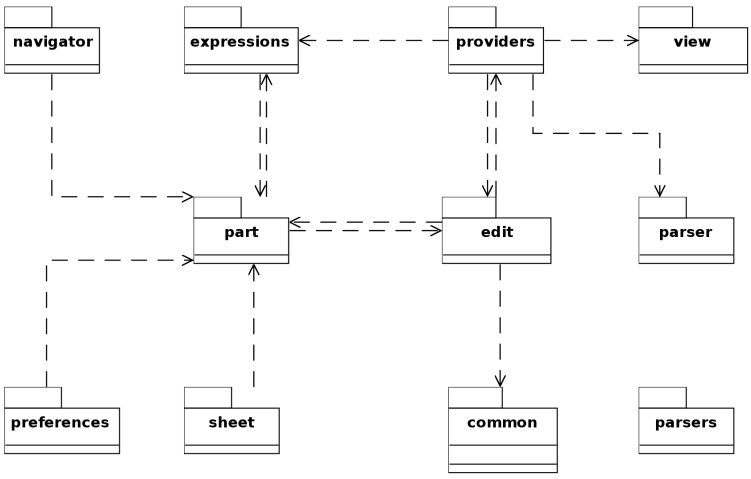


Abbildung 6.17: Pakete des Zustandsdiagrammeditors

6.3.3.1 Struktur

Die Komponente besteht aus drei Teilkomponenten: **MQP-Metamodell**, **MQP-GUI** und **MQP-Measurement** (siehe Abbildung 6.18).

MQP-Metamodell (de.upb.pg.rmc.mqp): Dies ist eine Realisierung des MQP-Metamodells. Die Teilkomponente stellt anderen Teilkomponenten das Interface **IMQPInternalModel** zur Verfügung. Detail ist unter 6.3.3.2 aufgeführt.

MQP-GUI: Dieser Bestandteil von **ModelQualityPlan** umfasst weitere vier Teilkomponenten, die die Benutzeroberfläche zur Erzeugung und Pflege von MQPs zur Verfügung stellt. Die einzelnen Teilkomponenten sind unter 6.3.3.3 aufgeführt.

MQP-Measurement (de.upb.pg.rmc.measurement): Die Komponente **MQP-Measurement** ist für die Durchführung von Messungen des MQP auf einem UMLZ-Modell zuständig. Details sind unter 6.3.3.4 aufgeführt.

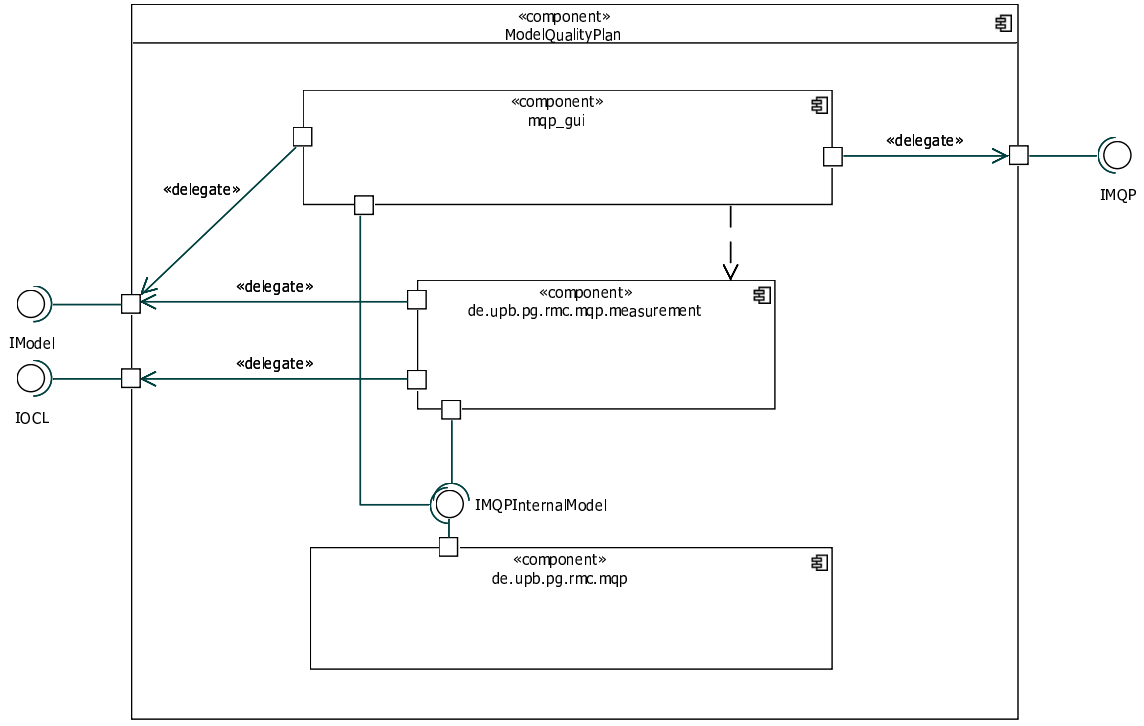


Abbildung 6.18: Komponentendiagramm: MQP

6.3.3.2 Metamodell

Das Metamodell des Modellqualitätsplans (kurz MQP-Metamodell) wurde von Hendrik Voigt entwickelt. Das im Weiteren beschriebene Meta-MQP-Modell wurde verfeinert und mit dem Eclipse Modelling Framework (EMF) [EMF] modelliert. Das Meta-MQP-Modell

enthält fünf Teilmodelle: **Kontextmodell** (engl. ContextModel), **Informationsbedürfnismodell** (engl. Information Need Model), **Qualitätsmodell** (engl. Quality Model), **Messmodell** (engl. Measurement Model) und **Präsentationsmodell** (engl. Persentation Model). Folgendes Paketdiagramm 6.19 stellt Zusammenhänge zwischen einzelnen Modellen dar.

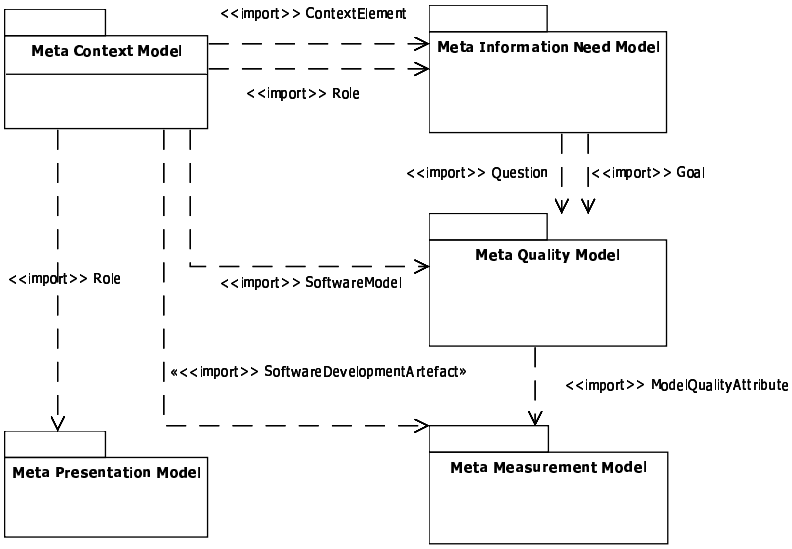


Abbildung 6.19: Paketdiagramm: Metamodell des Modellqualitätsplans

Kontextmodell

Das Kontextmodell beschreibt das Umfeld, in dem ein Software-Modell betrachtet wird. Eine detaillierte Beschreibung ist in [VE08] angegeben. Abbildung 6.20 stellt das Klassendiagramm des Kontextmodells dar.

Informationsbedürfnismodell

Das Informationsbedürfnismodell soll gewährleisten, dass ausschließlich Aspekte eines Softwaremodells betrachtet werden, die in dem vom Kontextmodell gegebenen Umfeld wichtig erscheinen [VE08]. Abbidlung 6.21 stellt das Klassendiagramm des Informationsbedürfnismodells dar.

Qualitätsmodell

Das Qualitätsmodell stellt einen weit verbreiten Ansatz dar, Qualität nach dem Divide-and-Conquer-Prinzip in mehrere Bestandteile zu gliedern und auf diese Weise zu präzisieren [VE08]. Abbildung 6.22 stellt das Klassendiagramm des Qualitätsmodells dar.

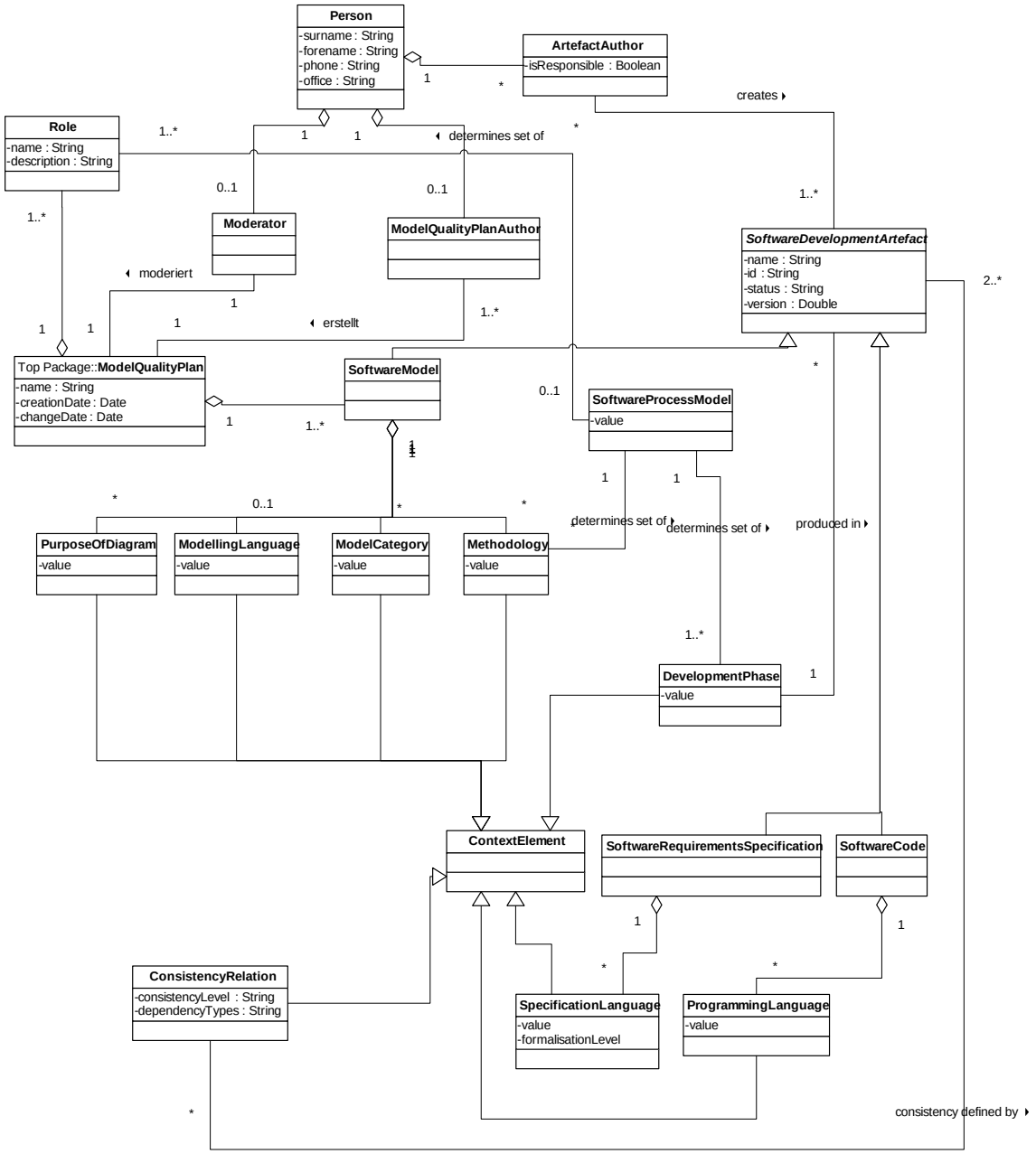


Abbildung 6.20: Klassendiagramm: Meta-Kontextmodell

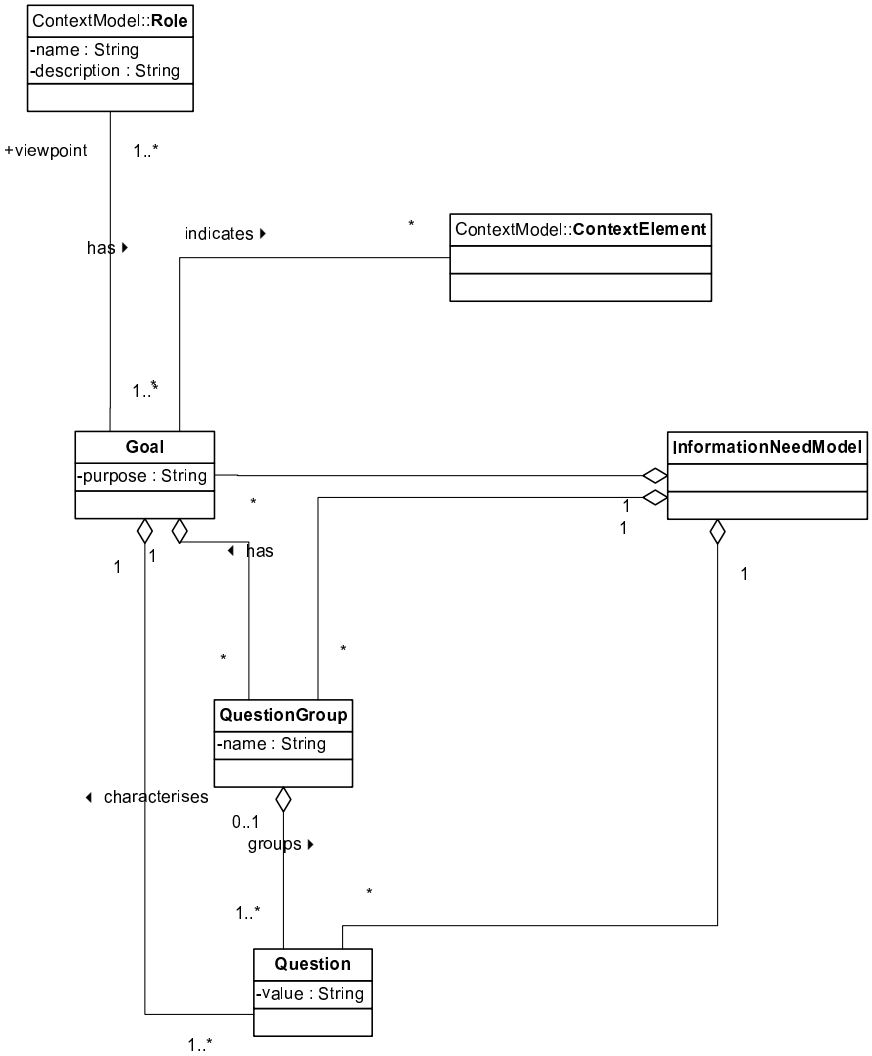


Abbildung 6.21: Klassendiagramm: Meta-Informationsbedürfnismodell

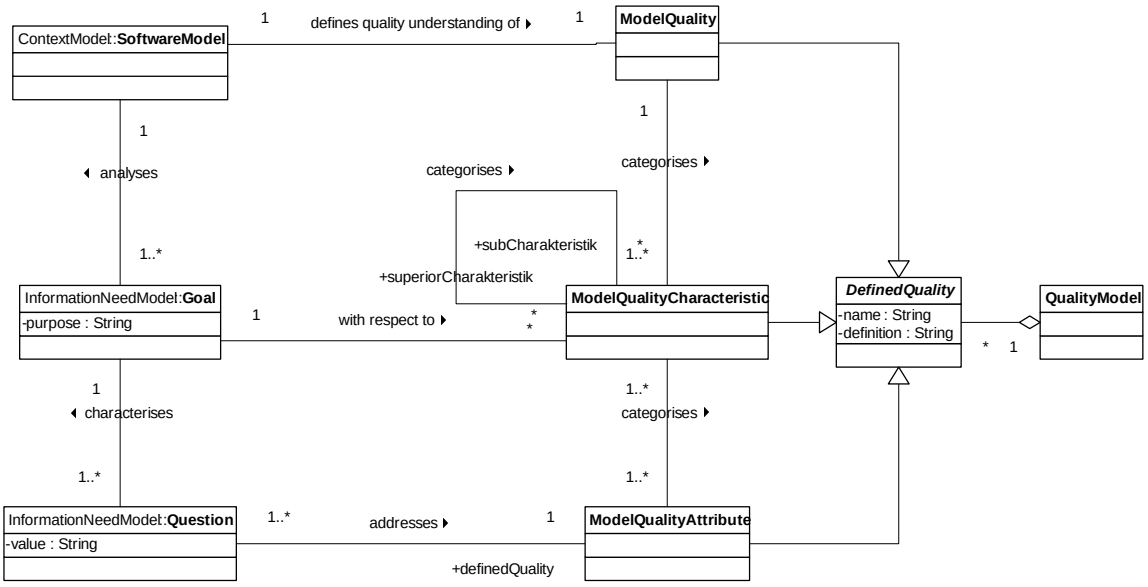


Abbildung 6.22: Klassendiagramm: Meta-Qualitätsmodell

Messmodell

Das Messmodell beschreibt alle notwendigen Messungen und Analysen, die in der Qualitätsprüfung durchgeführt werden sollen. Das Messmodell beantwortet folgende Fragen:

- Was wird gemessen?
- Wie wird gemessen?
- Welche Messungen existieren?
- Wie werden Messungen bestimmt?
- Wie hängen diese Messungen mit dem Qualitätsmodell zusammen?

Eine detaillierte Beschreibung ist in [VE08] zu finden.

Abbildung 6.23 stellt das Klassendiagramm des Messmodells dar. Die Klassen `ObjectiveBaseMeasure` und `DerivedMeasure` haben jeweils das Attribut `OCLExpression`, in dem eine OCL-Anfrage gespeichert wird. Die Klasse `Indicator` enthält die Attribute `calculation` und `calculationValue`. Diese definieren, wie ein `Indicator` durch enthaltene Teilmessungen (`BaseMeasure` oder/und `DerivedMeasure`) bewertet wird. Die Klasse `AuxiliaryFunctions` (oben rechts in Abbildung 6.23) definiert eine externe OCL-Funktion für Elemente eines UML/Z-Modells und dient der Vereinfachung der Definitionen von OCL-Anfragen in Messungen.

Die Klassen, die mit dem Stereotype **config** gekennzeichnet sind, werden ausschließlich in der Konfigurationsdatei des MQP genutzt. Im eigentlichen Modell verweisen die Klassen, deren Klassenname mit **Link** endet (wie z.B. `MeasureLink`), anhand des **acronym**-Attributs auf die entsprechenden Einträge in der Konfigurationsdatei.

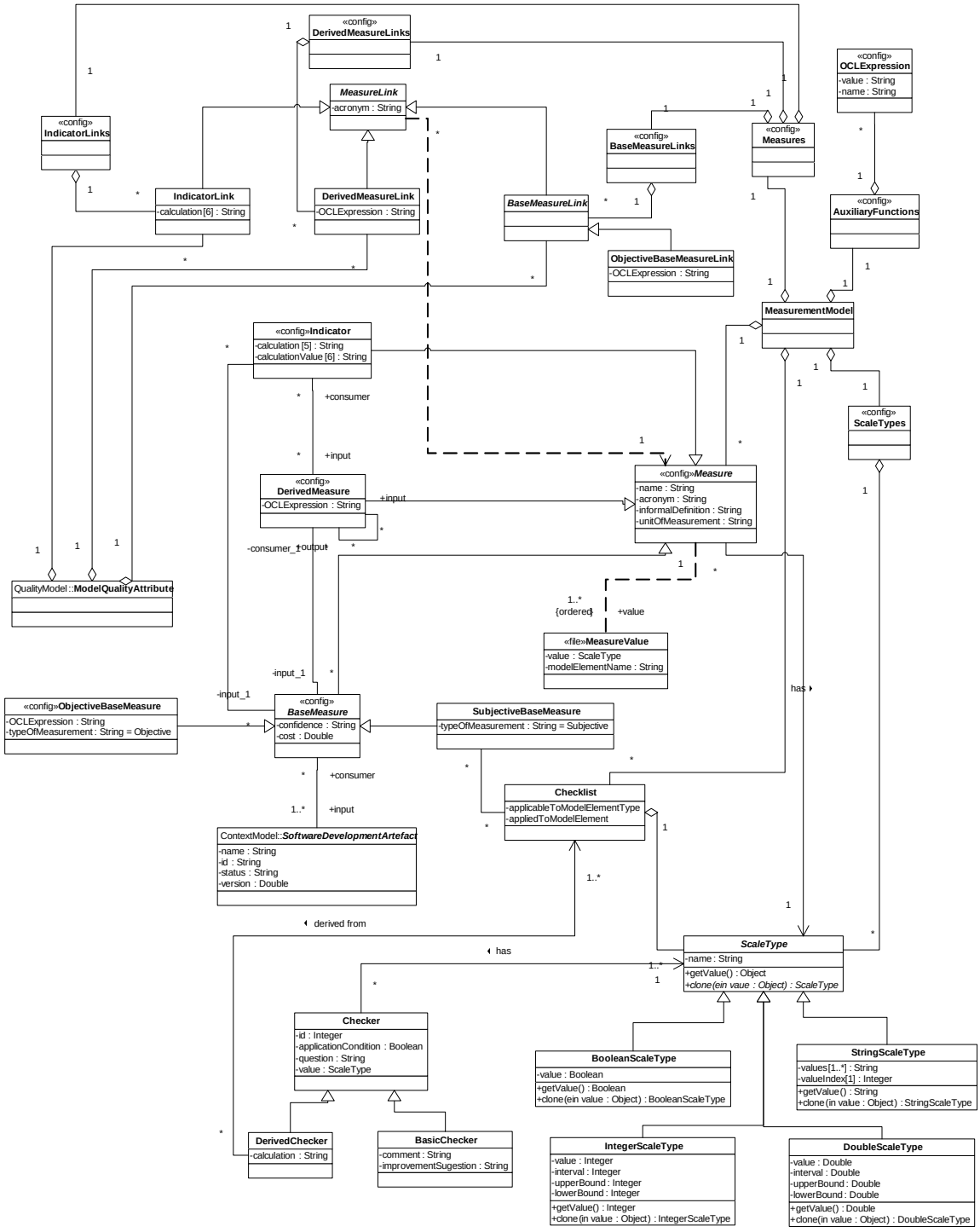

«config» DerivedMeasureLinks
MeasureLink -acronym : String
«config» IndicatorLinks
IndicatorLink -calculation [6] : String
DerivedMeasureLink -OCLExpression : String
BaseMeasureLink
«config» BaseMeasureLinks
«config» Measures
«config» OCLExpression -value : String -name : String
«config» AuxiliaryFunctions
ObjectiveBaseMeasureLink -OCLExpression : String
MeasurementModel
«config»Indicator -calculation [5] : String -calculationValue [6] : String
«config» ScaleTypes
«config» DerivedMeasure -OCLExpression : String
«config»Measure -name : String -acronym : String -informalDefinition : String -unitOfMeasurement : String
QualityModel::ModelQualityAttribute
«file»MeasureValue -value : ScaleType -modelElementName : String
«config»ObjectiveBaseMeasure -OCLExpression : String -typeOfMeasurement : String = Objective
«config» BaseMeasure -confidence : String -cost : Double
SubjectiveBaseMeasure -typeOfMeasurement : String = Subjective
Checklist -applicableToModelElementType -appliedToModelElement
ContextModel::SoftwareDevelopmentArtefact -name : String -id : String -status : String -version : Double
ScaleType -name : String +getValue() : Object +clone(ein vaue : Object) : ScaleType
Checker -id : Integer -applicationCondition : Boolean -question : String -value : ScaleType
BooleanScaleType -value : Boolean +getValue() : Boolean +clone(ein value : Object) : BooleanScaleType
StringScaleType -values[1..*] : String -valueIndex[1] : Integer +getValue() : String +clone(in value : Object) : StringScaleType
DerivedChecker -calculation : String
BasicChecker -comment : String -improvementSugestion : String
IntegerScaleType -value : Integer -interval : Integer -upperBound : Integer -lowerBound : Integer +getValue() : Integer +clone(in value : Object) : IntegerScaleType
DoubleScaleType -value : Double -interval : Double -upperBound : Double -lowerBound : Double +getValue() : Double +clone(in value : Object) : DoubleScaleType
+consumer
+input
+output
-consumer_1
-input_1
1..*
{ordered}
+value
has
derived from

Abbildung 6.23: Klassendiagramm: Meta-Messmodell

Präsentationsmodell

Das Präsentationsmodell bereitet Information zur Vorstellung der Ergebnisse auf. Abbildung 6.24 stellt das Klassendiagramm des Präsentationsmodells dar.

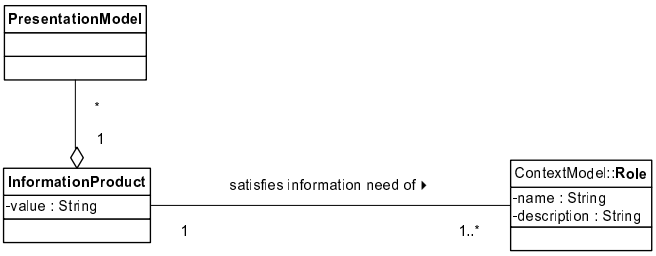


Abbildung 6.24: Klassendiagramm: Meta-Präsentationsmodell

6.3.3.3 Teilkomponente MQP-GUI

Die Teilkomponente **MQP-GUI** erfüllt folgende Aufgaben:

- stufenweises Anlegen eines Modellqualitätsplans
- Anstoßen einer Messung auf einem UML/Z-Modell
- Anstoßen der Diagnose (vergleiche Beschreibung der Komponente **Diagnostics** in Kapitel 6.3.2)
- Präsentation der Messergebnisse

Durch die Festlegung auf die Eclipse-Plattform im UML/Z-Modell und im UML/Z-Editor folgt, dass auch die MQP-GUI Komponente auf Eclipse basiert. Der von EMF für das MQP-Metamodell generierte Baumeditor aus dem Paket mqp.editor wurde als Grundlage für die MQP-Benutzeroberfläche verwendet und um weitere Reiter ergänzt. Für jedes MQP-Teilmodell wurde ein Reiter mit einem Formular zum Bearbeiten dieses Teilmodells erstellt.

Die Formulare basieren auf den üblichen Eclipse-Bibliotheken für Benutzeroberflächen: **SWT**, **Eclipse Forms** und **JFace**. Lediglich für die Präsentation der Ergebnisse wurde eine Swing-Komponente verwendet, da dafür keine geeignete SWT-Komponente verfügbar war. Details dazu stehen in der Entwicklerdokumentation auf Seite 197. Die Formulare sind aufgeteilt auf die Pakete `mqp.ui`, `mqp.ui.viewmodel` und `mqp.ui.commands`. Diese Unterteilung folgt aus der Verwendung des **Model-View-ViewModel-Patterns**, welches in der Entwicklerdokumentation auf Seite 197 näher beschrieben wird.

Das Paket mqp.edit enthält Hilfsklassen (`LabelProvider` und `ItemProvider`) für die Darstellung von Objekten aus dem MQP-Metamodell, z.B. in Tabellen und Baumeditoren.

Im Qualitätsmodell wurde zusätzlich ein mit GMF erstellter Diagrammeditor in den Editor integriert, dieser befindet sich im Paket mqp.diagram.quality. Dafür waren umfangreiche Änderungen am Editor erforderlich, die in der Entwicklerdokumentation auf Seite 197 beschrieben werden.

Das Komponentendiagramm in Abbildung 6.25 zeigt den Zusammenhang zwischen den Paketen der MQP-Benutzeroberfläche.

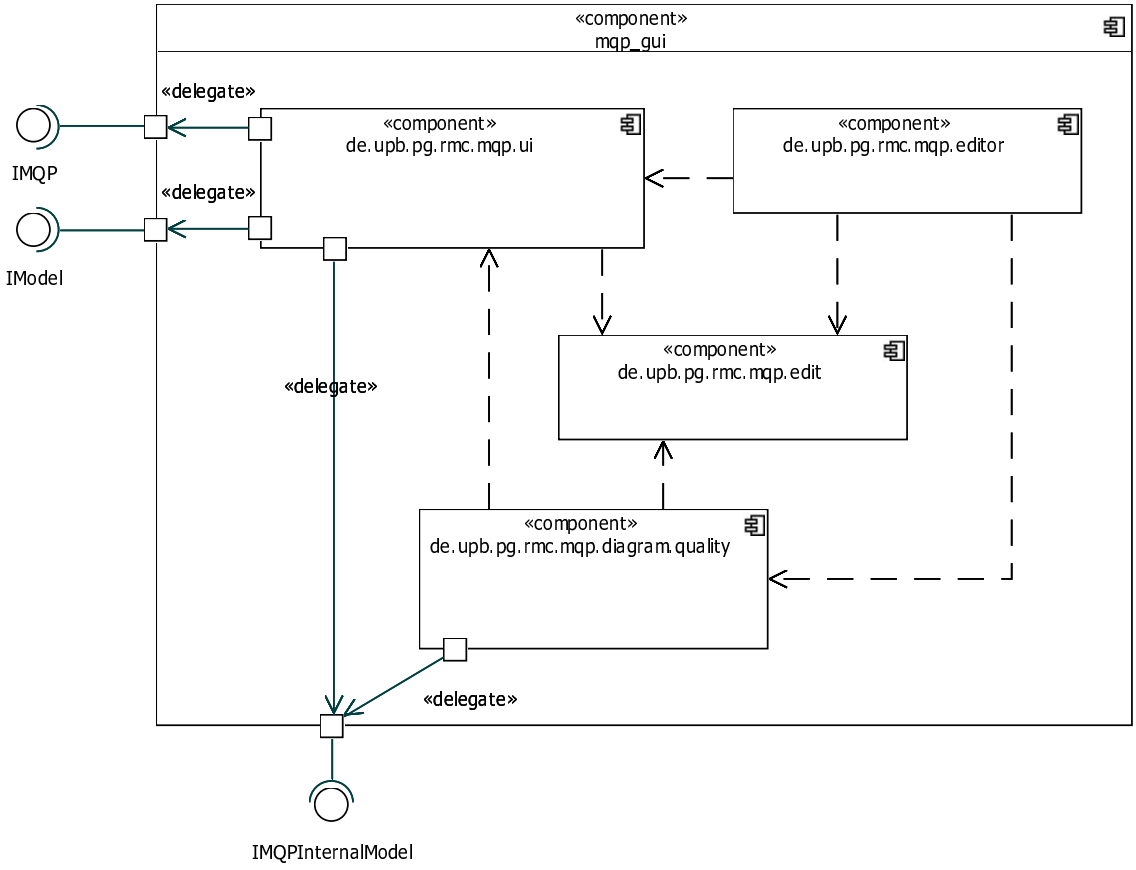


Abbildung 6.25: Komponentendiagramm: MQP-GUI

6.3.3.4 Teilkomponente MQP-Measurement

Die Teilkomponente **MQP-Measurement** ist für Durchführungen von Messungen auf UML/Z-Modellen zuständig. Das Vorgehen beim Messen unterliegt nachstehender Reihenfolge:

- **MQP-GUI** startet einer Messung (engl. Measurement) und übergibt der Teilkomponente **MQP-Measurement** das zu messende UML/Z-Modell.
- Die genutzten bzw. ausgewählten Messungen werden über die Schnittstelle **IMQPInternalModel** ermittelt.
- Die Messungen der Typen `ObjectiveBaseMeasure` und `DerivedMeasure` werden mit darin definierten OCL-Ausdrücken (engl. OCL-Expression) über die Schnittstelle **IOCL** (Kapitel 6.2.4) in der Komponente **OCLWrapper** durchgeführt (vergleiche Kapitel 6.3.6).

- Die Messungen vom Typ `Indicator` werden basierend auf den Ergebnissen der `ObjectiveBaseMeasure` und `DerivedMeasure` berechnet.
- Die Messergebnisse werden für die Präsentation in **MQP-GUI** und für die Durchführung der Diagnose in der Komponente **Diagnostics** aufbereitet.
- Außerdem werden die Messergebnisse serialisiert als XML-Datei gespeichert. Dadurch ist es möglich, die Ergebnisse mehrerer Messungen zu speichern.

Das Kompositionsstrukturdiagramm aus Abbildung 6.26 zeigt die Relationen zwischen relevanten Klassen und Schnittstellen der Komponente.

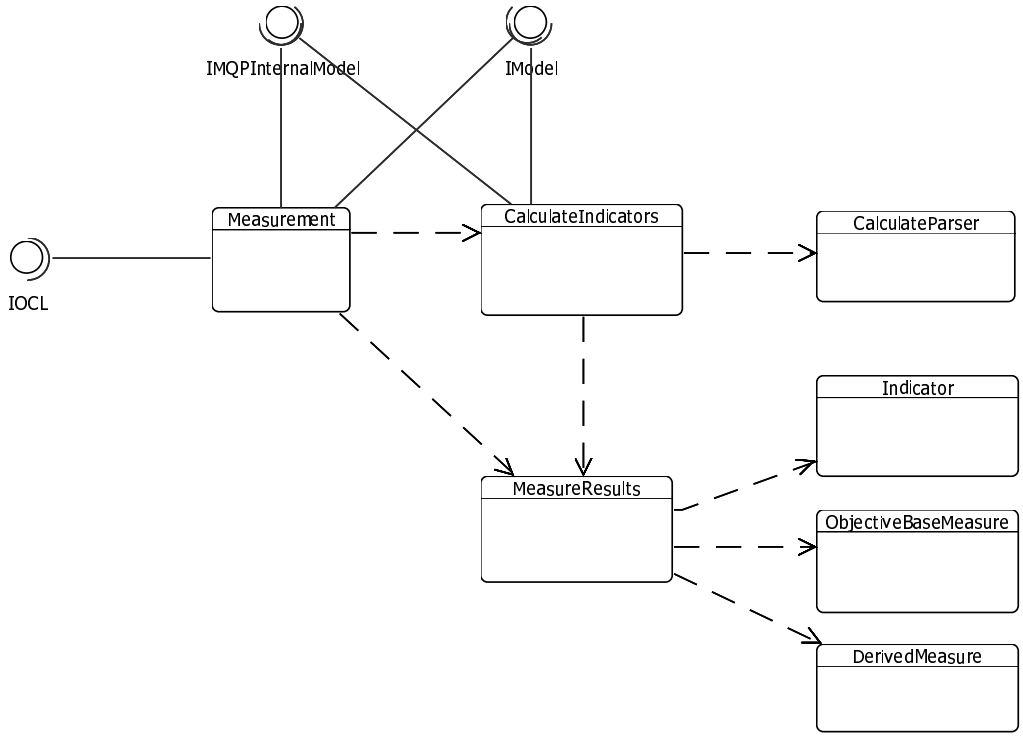


Abbildung 6.26: Kompositionsstrukturdiagramm: MQP-Measurement

Abbildung 6.27 zeigt das Klassendiagramm der Komponente **MQP-Measurement**. Die Klasse `Measurement` ist eine zentrale Kontrollklasse der Komponente. Die Klasse `Measurement` fragt über die Schnittstelle **IMQPInternalModel** der Komponente **MQP-Metamodel** (`de.upb.gp.rmc.mqp`) die genutzten bzw. gewählten Messungen an, übergibt der Komponente **OCLWrapper** den OCL-Ausdruck der Messungen der Typen `ObjectiveBaseMeasure` und `DerivedMeasure` und das zu messende UML/Z-Modell über die Schnittstelle **IOCL**. Die Messungen werden im **OCLWrapper** auf dem UML/Z-Modell durchgeführt und die Ergebnisse der Messungen werden der Klasse `Measurement` zurückgeliefert. Die Klasse `Measurement` schreibt die Ergebnisse der `ObjectiveBaseMeasures` und `DerivedMeasures` und speichert die Messungen in der Klasse `MeasurementResult`. Die Referenz der `MeasurementResults` und der zu messenden `Indicators` wird der Klasse `CalculateIndicators` übermittelt. Die Klasse `CalculateIndicators` berechnet die Ergebnisse der Indikatoren auf Basis der Ergebnisse der `ObjectiveBaseMeasures` und `DerivedMeasures`.

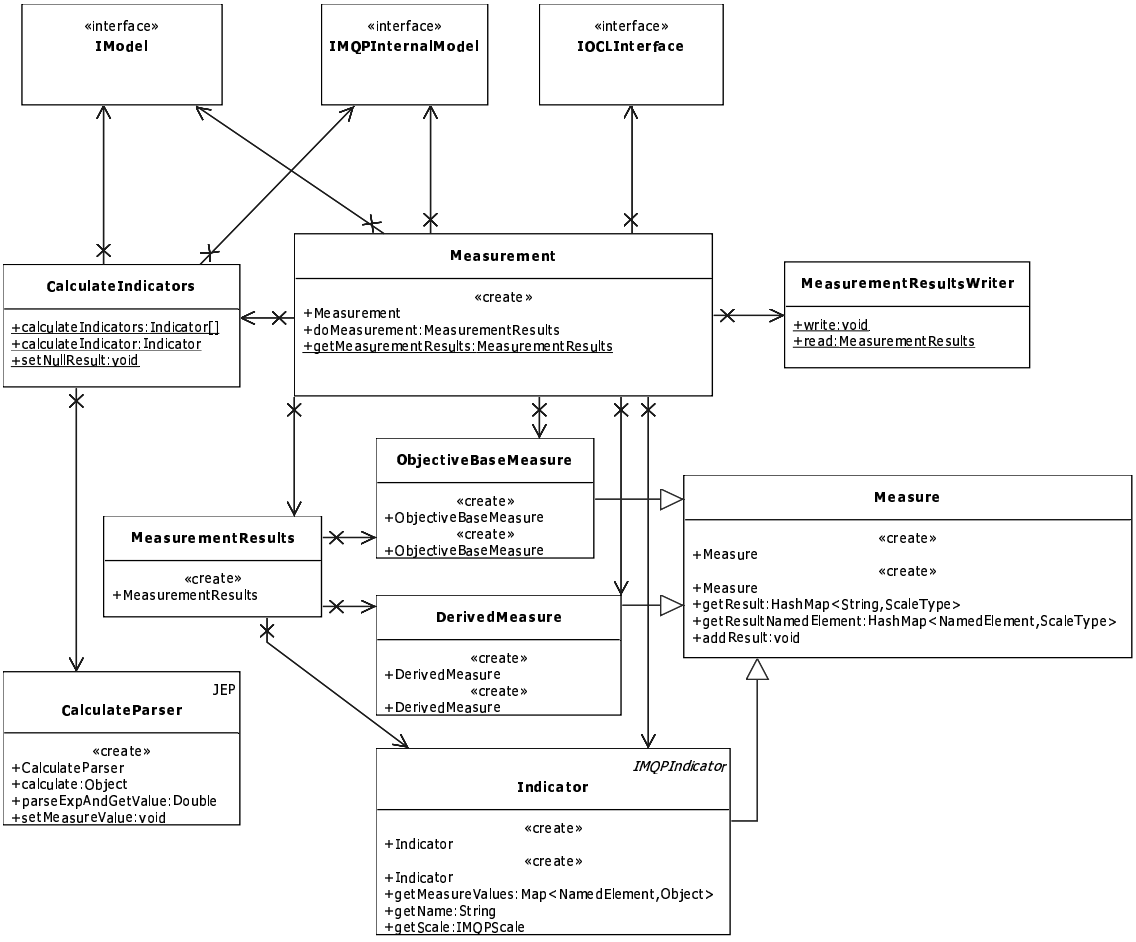


Abbildung 6.27: Klassendiagramm: MQP-Measurement

Das Messergebnis in der Klasse `MeasurementResults` wird am Ende durch die Klasse `MeasurementResultsWriter` serialisiert als XML-Datei gespeichert.Abbildung 6.27 zeigt das Klassendiagramm der Komponente **MQP-Measurement**. Die Klasse `Measurement` ist eine zentrale Kontrollklasse der Komponente. Die Klasse `Measurement` fragt über die Schnittstelle **IMQPInternalModel** der Komponente **MQP-Metamodel** (`de.upb.gp.rmc.mqp`) die genutzten bzw. gewählten Messungen an. Weiterhin übergibt `Measurement` der Komponente **OCLWrapper** den OCL-Ausdruck der Messungen der Typen `ObjectiveBaseMeasure` und `DerivedMeasure` und das zu messende UML/Z-Modell über die Schnittstelle **IOCL**. Die Messungen werden im **OCLWrapper** auf dem UML/Z-Modell durchgeführt und die Ergebnisse der Messungen werden der Klasse `Measurement` zurückgeliefert. Die Klasse `Measurement` schreibt die Ergebnisse der `ObjectiveBaseMeasures` und `DerivedMeasures` und speichert die Messungen in der Klasse `MeasurementResult`. Die Referenz der `MeasurementResults` und der zu messenden `Indicators` wird der Klasse `CalculateIndicators` übermittelt. Die Klasse `CalculateIndicators` berechnet die Ergebnisse der Indikatoren auf Basis der Ergebnisse der `ObjectiveBaseMeasures` und `DerivedMeasures`. Das Messergebnis in der Klasse `MeasurementResults` wird am Ende durch die Klasse `MeasurementResultsWriter` serialisiert als XML-Datei gespeichert.

6.3.4 Diagnostics

Im folgenden Abschnitt wird auf die Architektur der **Diagnostics**-Komponente eingegangen. Hierbei handelt es sich um eine Komponente von RMC, die eine Menge von Indikatoren entsprechend der Definition aus Abschnitt 6.3.3 analysiert. Nach einer vorgegebenen Konfiguration ermittelt die Diagnose also Verbesserungsvorschläge im Hinblick auf gegebene Modellmängel.

Die Diagnose stellt dabei eine Verbindung der **ModelQualityPlan**-Komponente (Abschnitt 6.3.3) und der **Refactoring**-Komponente (Abschnitt 6.3.5) dar. Dabei erhält die Diagnose aus der **ModelQualityPlan**-Komponente Messergebnisse als Indikatoren und wertet diese dann in Form von Diagnosen und Therapien aus. Mit den gefundenen Therapien (**Therapiekandidaten**) kann der Anwender, wenn es sich um eine **Texttherapie** handelt, einer angezeigten Anleitung zur Modellverbesserung folgen. Eine weitere Option ist eine Modellverbesserung mit Hilfe der Refactoringkomponente durchzuführen.

Im Folgenden wird nun auf die Struktur der Diagnose wie auch auf ausgewählte Verhaltensabläufe näher eingegangen.

6.3.4.1 Struktur

Abbildung 6.28 zeigt die wichtigsten Klassen der **Diagnostics**-Komponente.

Mit Hilfe der Klassen `ConfigurationParser` und `ConfigurationFile` werden alle für die Diagnose benötigten Informationen aus der XML-Konfigurationsdatei eingelesen und für die spätere Verwendung gespeichert. Die Diagnose wird durch den Anwender im Präsentationsmodell der **ModelQualityPlan**-Komponente aufgerufen (Abschnitt 6.3.3).

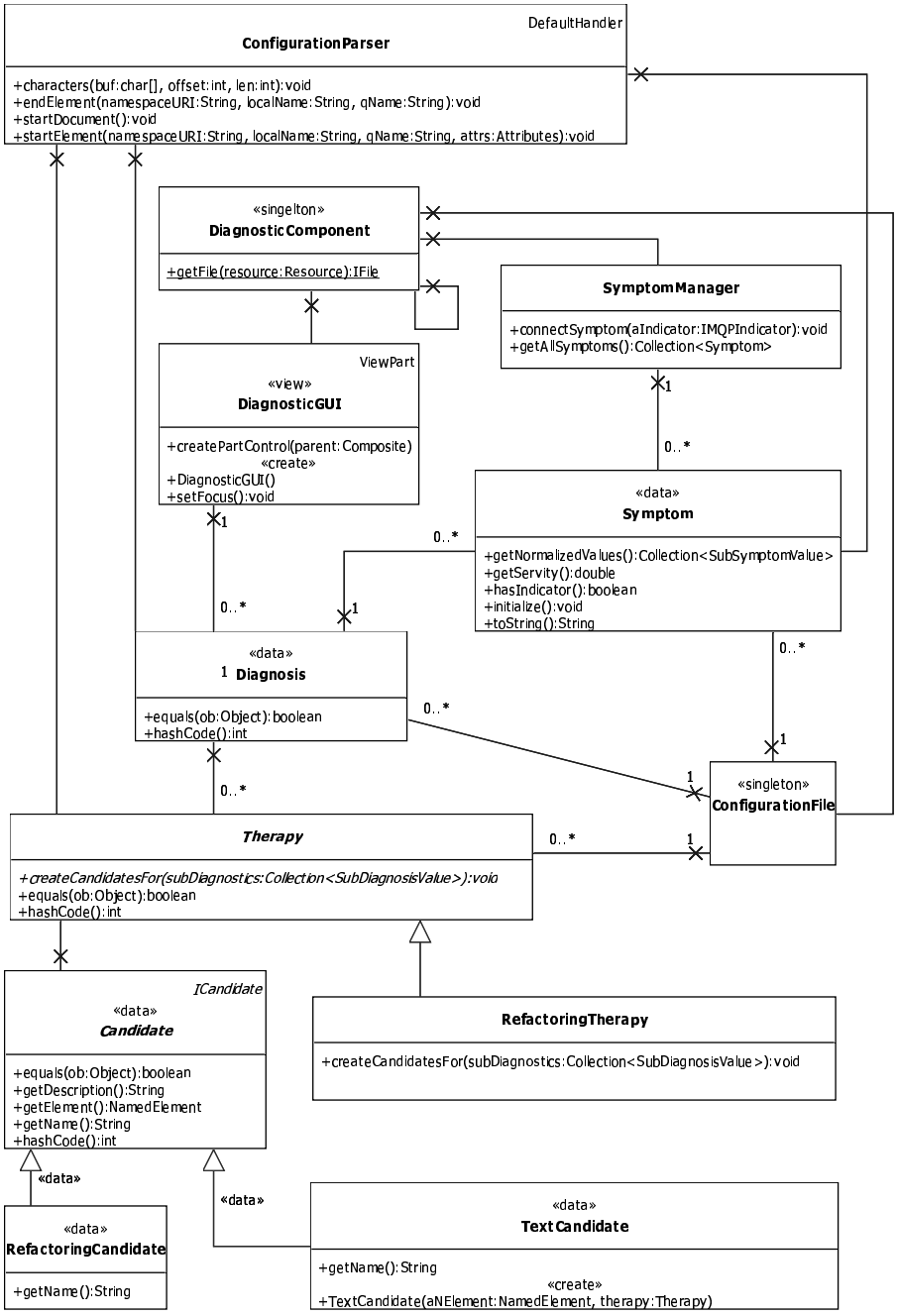


Abbildung 6.28: Klassendiagramm: Diagnostics-Komponente

Die Klasse `DiagnosticGUI` beinhaltet alle Informationen zur Darstellung der Diagnose. Bevor eine Diagnose angezeigt werden kann, wird für jeden Indikator bzw. jedes Modellelement, welches immer zu einem Indikator gehört, das Messergebnis aus dem MQP durch Aufruf des Normalisierungsmanagers durch den Symptommanager (`SymptomManager`) normalisiert (siehe Seite 174). Ist der jeweilige normalisierte Wert größer oder gleich dem Grenzwert, so wird für diese Symptome eine passende Diagnose ermittelt (abhängig von den Informationen in der Konfigurationsdatei).

Wurde eine Diagnose ermittelt, so wird nach passenden Therapiekandidaten (`Candidate`) gesucht. Neben Textkandidaten (`TextCandidate`) gibt es in der derzeitigen Version der Diagnose auch Refactoringkandidaten (`RefactoringCandidate`) für die in der **Refactoring**-Komponente nach passenden Refactoringtherapien gesucht wird. Bei erfolgreicher Suche werden Therapien (`RefactoringTherapy`) zurück an die Diagnose zur Auswahl durch den Benutzer übergeben. Texttherapiekandidaten sind einfache textuelle Beschreibungen, welche in der GUI dem Benutzer Modellverbesserungen vorschlagen, die er selbst am Modell durchführen kann.

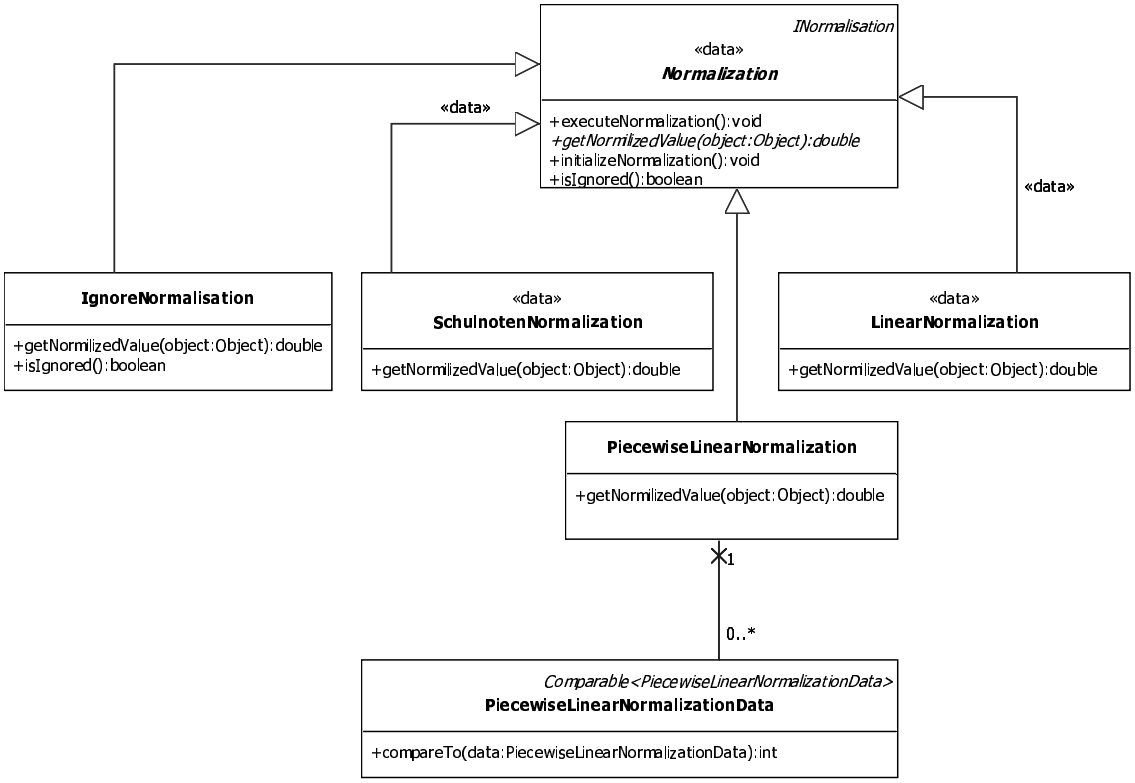


Abbildung 6.29: Klassendiagramm: Normalisierung

Das Klassendiagramm in Abbildung 6.29 zeigt die Klasse `Normalization` und die von dieser Klasse erbenden Klassen `IgnoreNormalization`, `SchulnotenNormalization`, `PiecewiseLinearNormalization` und `LinearNormalization`. Jedem Indikator wird im MQP eine **Skala** zugewiesen. Einige dieser im MQP bereits vordefinierten Skalen existieren in

der derzeitigen Version der Diagnose in Form von Normalisierungsklassen.

Wird eine neue Skala im MQP erstellt, so muss diese, sofern der damit gemessene und bewertete Indikator in der Diagnose bearbeitet werden soll, der Diagnose durch Hinzufügen einer neuen Normalisierungsklasse bekannt gemacht werden (siehe Kapitel III).

Bei der **Normalisierung** geht es darum, die einzelnen gemessenen und bewerteten Indikatoren für Modelmängel gegeneinander vergleichbar bzw. darstellbar zu machen. Dieses Ziel wird innerhalb der einzelnen Normalisierungstypklassen dadurch erreicht, dass Ergebnisse der Normalisierungen immer im Wertebereich zwischen -1 und 1 liegen müssen.

Ist ein Wert nach der Normalisierung außerhalb dieses Bereiches, so wird ein Fehler ausgelöst. Gründe hierfür können entweder falsche Hilfsvariablen und Werte aus der Konfigurationsdatei sein, oder die verwendete Normalisierung liefert ein nicht gültiges Ergebnis.

Die zu normalisierenden Ergebnisse werden an die entsprechenden Normalisierungsklassen übermittelt. Je nach Normalisierungsart ist dieses Objekt dann in ein entsprechendes Format der jeweiligen Normalisierungsklasse umzuwandeln. Können Werte nicht normalisiert werden, wird eine Fehlermeldung ausgegeben.

Im Folgenden werden die derzeitig unterstützten Normalisierungen kurz beschrieben:

SchulnotenNormalization: Hierbei handelt es sich um eine direkte Zuordnung eines Strings zu einem Wert zwischen -1 und 1, welcher zur Weiterverarbeitung ohne jegliche weitere Berechnung zurückgegeben wird. Listing 6.1 zeigt die Skala mit dem Namen HighLow vom Typ Fuzzy. Hier sind insgesammt fünf Strings jeweils genau einem Wert zugewiesen.

```
<normalization type="Fuzzy" name="HighLow">
  <parameter name="Very Low" value="0.1" />
  <parameter name="Low" value="0.2" />
  <parameter name="Middle" value="0.3" />
  <parameter name="High" value="0.5" />
  <parameter name="Very High" value="0.8" />
</normalization>
```

Listing 6.1: Beispiel für eine Schulnotennormalisation

LinearNormalization: Diese Klasse berechnet einen Wert mit Hilfe der Steigungsgleichung einer Geraden. Die Gerade kann sowohl monoton steigend als auch monoton fallend sein (siehe Abbildung 6.30). Für die Berechnung benötigte Hilfsvariablen (in Abbildung 6.30 mit a und b bezeichnet) werden aus der Konfigurationsdatei eingelesen bzw. in der Klasse selbst berechnet.

PiecewiseLinearNormalization: Bei der **PiecewiseLinearNormalization** handelt es sich um eine Normalisierung, die aus mehreren monoton steigenden und/oder monoton fallenden Streckenzügen bestehen kann (Beispiel in Abbildung 6.31), welche sich anhand von Koordinaten aus der Konfigurationdatei zusammensetzen lässt (Abbildung 6.31 zeigt beispielsweise die Hilfsvariablen a, b, c und d). Hierbei können einzelne Stücke zwischen zwei Koordinaten unterschiedliche Steigungen haben. Es wird berechnet, zwischen welchen zwei x-Werten bzw. in welchem Streckenabschnitt der zu

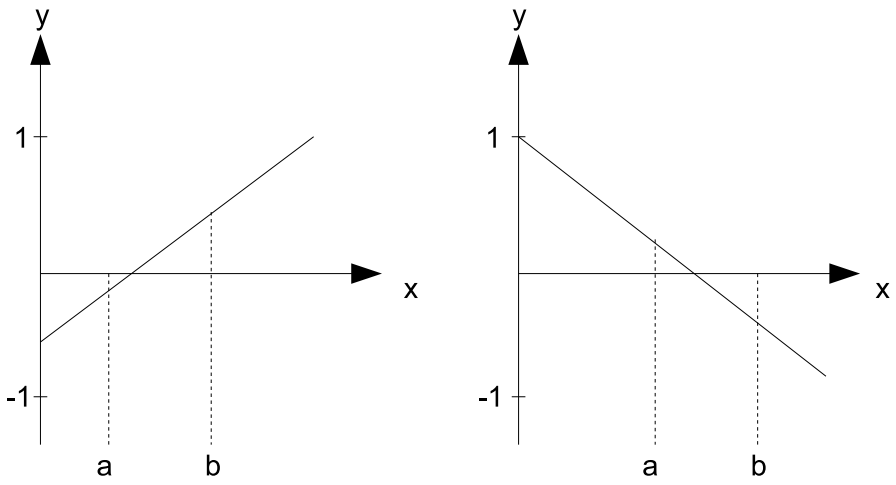


Abbildung 6.30: Beispiel: lineare Normalisierung

normalisierende Wert sich befindet, und dann wie bereits in der `LinearNormalization` weiter verfahren.

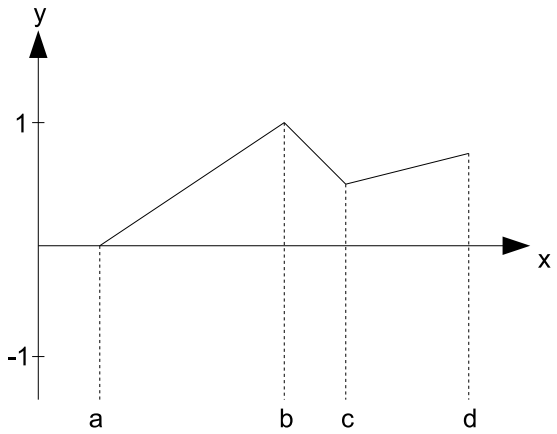


Abbildung 6.31: Beispiel: stückweise lineare Normalisierung

IgnoreNormalisation: Mit Hilfe dieser Klasse werden alle Normalisierungen bzw. Indikatoren mit der Diagnose derzeit nicht bekannten Skalen abgefangen.

6.3.4.2 Verhalten

Im Folgenden Abschnitt wird für die eben beschriebene Struktur der **Diagnostics**-Komponente anhand von drei Sequenzdiagrammen näher auf das Verhalten der Komponente eingegangen. Der Übersichtlichkeit halber handelt es sich bei diesen Diagrammen um stark abstrahierte Sequenzdiagramme, welche nur die wesentlichen Ideen vermitteln sollen.

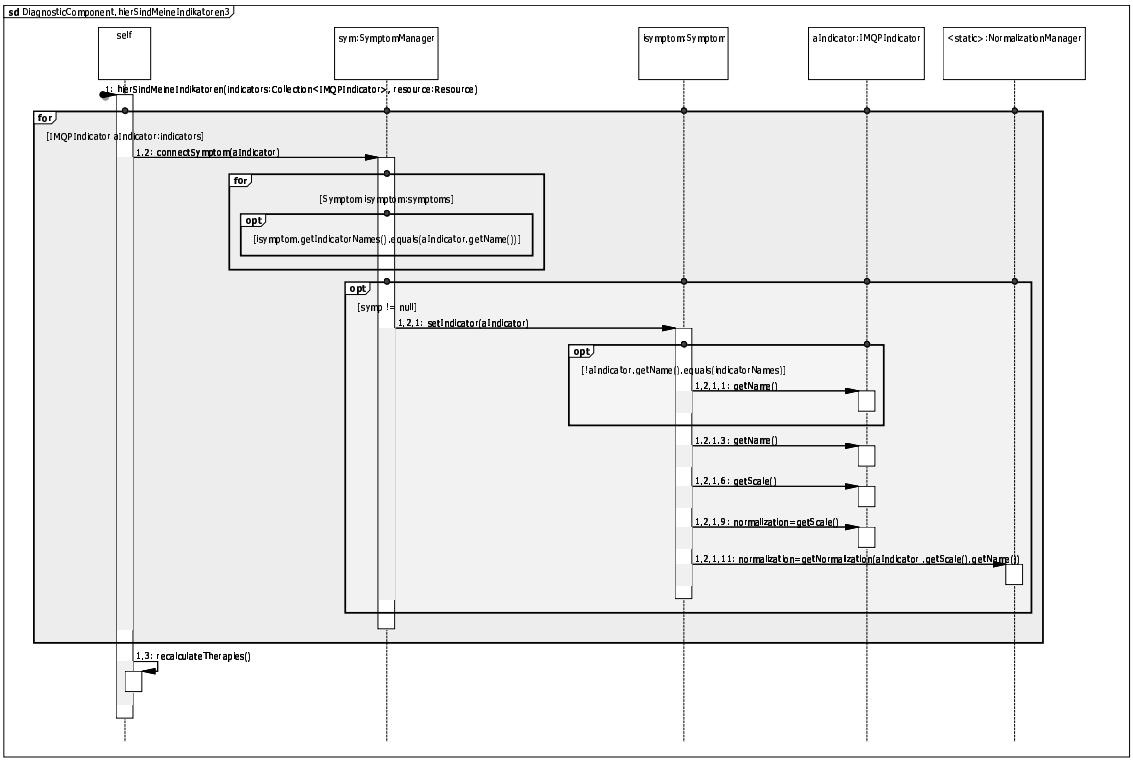


Abbildung 6.32: Sequenzdiagramm: Übergabe und Verarbeitung der übergebenen Indikatoren

Abbildung 6.32 zeigt den groben Ablauf, wie in der **Diagnostics**-Komponente nach der Übergabe einer Menge von Indikatoren (`hierSindMeineIndikatoren`) aus dem MQP im Symptommanager (`SymptomManager`) nach passenden Symptomen (`Symptom`) für die übergebenen Indikatoren gesucht wird. Existieren zu einem Indikator Symptome, so werden für jedes dieser Symptome Informationen aus dem Interface `IMQPIndicator` angefragt, welche im nächsten Schritt in der Normalisierung der übergebenen Messwerte aus dem MQP benötigt werden. Nachdem die Informationen erfolgreich geholt wurden, wird die Normalisierung (`NormalizationManager`) eingeleitet. Wurden für alle angegebenen Indikatoren Symptome gefunden bzw. zugeordnet[3] und die zugehörigen Normalisierungen erfolgreich durchgeführt, so können mit Hilfe von `recalculateTherapies`-Methode die gewonnenen Daten weiter verarbeitet werden, d.h. es wird nach passenden Diagnosen und zugehörigen Therapien gesucht.

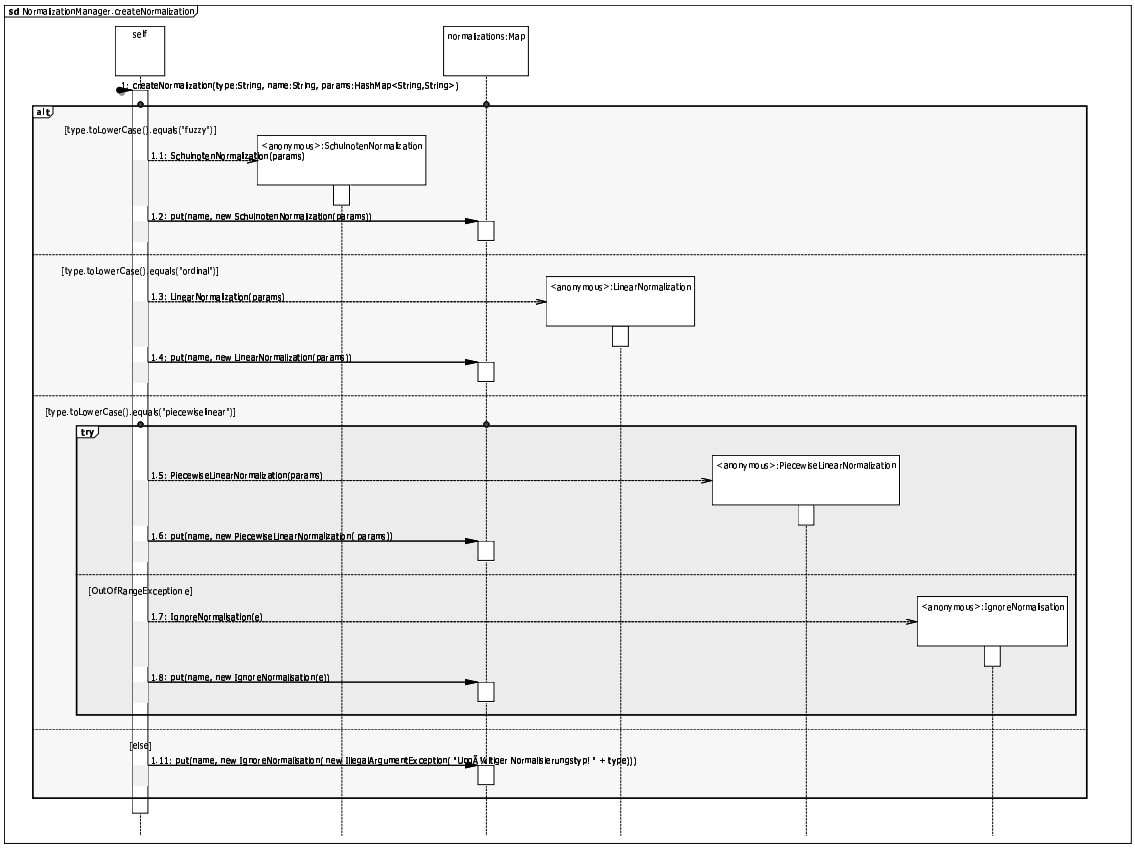


Abbildung 6.33: Sequenzdiagramm: Anzeige möglicher Normalisierungen

Die gerade angesprochene Normalisierung wird durch den Normalisierungsmanager (`NormalizationManager`) durchgeführt. Je nachdem, welche Skala einem Indikator bzw. einem Symptom zugewiesen wurde, wird die passende Normalisierung angestoßen. Im Sequenzdiagramm (Abbildung 6.33) sind die drei derzeit unterstützten Normalisierungen und deren zugehörige Normalisierungsklassen abgebildet. Ist einem Symptom eine nicht bekannte

[3] abhängig davon, ob die Indikatoren in der Konfigurationsdatei der Diagnose bekannt waren

Normalisierung zugeordnet, so wird diese Normalisierung nicht durchgeführt und eine Exception ausgelöst.

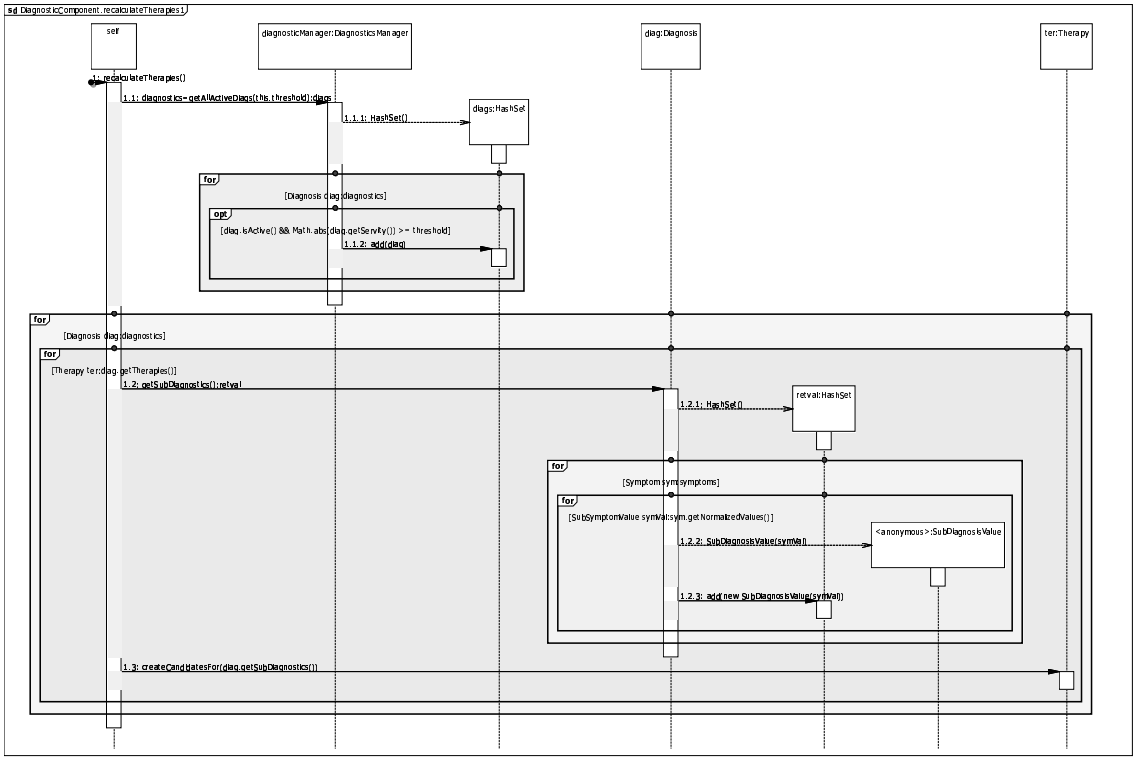


Abbildung 6.34: Sequenzdiagramm: Berechnung von Diagnosen und passender Therapien

Bei der Beschreibung des Sequenzdiagramms, welches in Abbildung 6.32 dargestellt ist, wurde bereits kurz auf die Methode `recalculateTherapies` Bezug genommen. Abbildung 6.34 zeigt nun den groben Ablauf dieser Methode auf, welche eine der wichtigsten Methoden der **Diagnostics**-Komponente darstellt.

Diese Methode wird einmal bei Ausführung der Diagnose aufgerufen und errechnet für den Grenzwert **threshold** aus der Konfigurationsdatei alle Diagnosen und zugehörigen Therapien[4] für die Anzeige in der GUI. Jedes Mal wenn der Grenzwert durch den Benutzer verändert wird, wird diese Methode erneut ausgeführt und die Berechnungen neu angestoßen. Abbildung 6.34 verdeutlicht, dass nach Ausführung der Methode `getAllActiveDiags` der `DiagnosticsManager` abhängig vom aktuellen Grenzwert eine Menge mit den aktuell passenden Diagnosen erstellt wird.

Für jede dieser Diagnosen werden dann `SubDiagnoseValues` erstellt, da einem Indikator mehrere Modellelemente mit eigenen gemessenen Indikatorwerten zugeordnet werden. Diese werden abhängig vom Grenzwert in der Diagnose-GUI angezeigt.

Durch Aufrufen der Methode `createCandidatesFor` mit den gerade berechneten `SubDiagnoseValue` als Parameter, werden nun passende Therapiekandidaten für die Diagnosen

[4] sowohl Text als auch Refactoringtherapien

gesucht.

Mit diesem Abschnitt, sollte ein Überblick über die Struktur der Diagnose gegeben werden. Weiter wurden für ausgesuchte Abläufe in Form von Sequenzdiagrammen auf das Verhalten der Diagnose näher eingegangen. Im Folgenden Abschnitt wird nun die Architektur Refactoringkomponente beschrieben.

6.3.5 Refactoring

Die **Refactoring**-Komponente führt verhaltenserhaltende Änderungen auf UML/Z-Modellen durch. Die Menge der Refactorings wird aus einer Konfigurationsdatei eingelesen, mit deren Hilfe einzelne Abfragen und atomare Modelländerungen zur Durchführung von Refactorings beeinflusst werden können, die im Zusammenspiel eine *verhaltenserhaltende* Modifikation eines UML/Z-Modells ergeben. Anweisungen und Anfragen an ein UML/Z-Modell sind in der Anfragesprache Object Constraint Language (OCL) formuliert und werden über die RMC-Komponente **OCLWrapper** auf einem Modell ausgeführt (siehe Abschnitt 6.3.6).

Die diesem Abschnitt zugrundeliegende Komponente von RMC wird aus der **UML/Z-Editor**-Komponente und der **Diagnostics**-Komponente aufgerufen. Dabei werden jeweils zu einer Menge von Modellelementen Refactorings gesucht.

Theoretische Ausarbeitungen zu Refactorings auf objekteorientierten Spezifikationsformalismen unterstützt von einer verhaltensbeschreibenden Prozessalgebra sind [RW07] zu entnehmen. Wie in Kapitel 2 beschrieben, entsprechen im Fall der Spezifikationssprache UML/Z Klassendiagramme unterstützt von Z-Annotationen objekteorientierten Spezifikationsformalismen, Zustandsdiagramme übernehmen die Rolle von Prozessalgebren. In Abschnitt 6.3.5.1 beschriebene Strukturen stimmen im Wesentlichen mit in [RW07] definierten Bestandteilen eines Refactorings überein. Abschnitt 6.3.5.2 geht auf interne Abläufe der **Refactoring**-Komponente ein.

6.3.5.1 Struktur

Zentrale Elemente des Klassendiagramms aus Abbildung 6.35 sind `RefactoringType` und `AbstractRefactoring`. `RefactoringType`-Objekte stellen einen Refactoringtyp dar, der unabhängig von konkreten Modellelementen spezifiziert ist. Dieser enthält sowohl die Bedingung (`ApplicabilityCondition`), bei der ein `RefactoringType` abhängig von einer Eingabemenge von Modellelementen ausführbar ist, als auch `Decisions`. Refactoringtypen werden beim Einlesen einer Konfigurationsdatei (`Parser`) erzeugt.

`RefactoringElement`-Objekte (nach außen durch das `IRefactoringElement` repräsentiert) stellen ausführbare Strukturen auf bestimmte Modellelementmengen dar.
Beim Anstoßen eines `IRefactoringElement`-Objektes von außen (Ausführen eines Refactorings auf bestimmte Modellelemente) wird mit Hilfe der verfügbaren Modellelemente eine `VariableToElementMap` erstellt. Diese beinhaltet `Variable`-Objekte, die durch beliebige Objekte belegbar sind.

Ist die `ApplicabilityCondition` validiert und alle `Decisions` getroffen, so kann mit der resultierenden Belegung von Variablen die Refactoringberechnung (`AbstractRefactring`) angestoßen werden. Hier werden in weiteren Bestandteilen des `AbstractRefactoring` (`Precondition, BeforeTemplate, Calculation, AfterTemplate`) mit Hilfe von OCL-Abfragen in Form von `IOCLQuery`-Objekten auf dem Modell ausgeführt. Diese Queries können Objekte als Ergebnis zurückliefern.

Der `RefactoringManager` implementiert sowohl das Interface `IRefactoringEditor` (siehe Abschnitt 6.2.2) als auch `IRefactoringDiagnostics` (siehe Abschnitt 6.2.3).

Detaillierte Informationen zu einzelnen Klassen sind in folgenden Abschnitten gegeben.

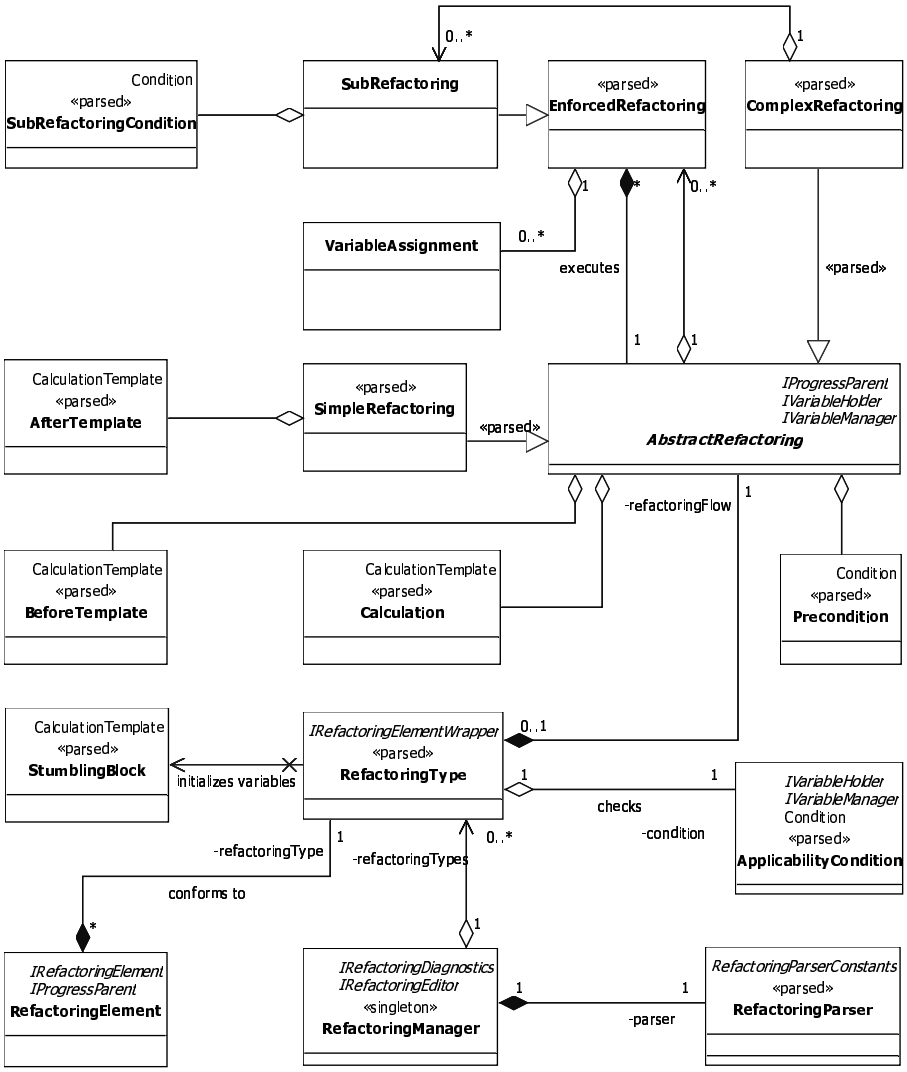


Abbildung 6.35: Klassendiagramm: Refactoring-Komponente

Die Klasse `Condition` in Abbildung 6.36 ist die Superklasse von Klassen, die eine Bedingung für beliebige Teilabläufe eines Refactorings ausdrücken. Das Überprüfen derartiger Bedingungen wird mit **true** oder **false** beantwortet und entscheidet über den weiteren

Verlauf eines Refactorings (vergleiche Abschnitt 6.3.5.2). Von `Condition` erbende Klassen erfüllen folgende Funktionalität:

ApplicabilityCondition: Objekte dieser Klasse stellen Bedingungen für die Ausführung eines Refactorings bei gegebener Modellelementmenge dar. Ist der Rückgabewert **false**, so werden keine weiteren Schritte des Refactoringablaufs durchgeführt. Die Bedingung wird bereits im Refactoringtyp überprüft und gilt damit für den ganzen Ablauf, der aus mehreren einzelnen `AbstractRefactoring`-Objekten bestehen kann. Eine `ApplicabilityCondition` besteht in den meisten Fällen aus genau einer Instanz von `IContextQuery`, so dass diese möglichst schnell validiert werden kann.

DecisionValidation: Mit Hilfe der `DecisionValidation` werden Entscheidungen des Anwenders (vergleiche Beschreibung zu Abbildung 6.38) auf bestimmte Bedingungen hin überprüft. Können diese Bedingungen, die mittels `OCLQuerys` definiert sind, nicht validiert werden, erhält der Anwender eine Beschreibung, die die Bedingung beschreibt und den Fehler der Entscheidungsangabe ausweist.

Precondition: Im Gegensatz zu `ApplicabilityCondition` definiert eine `Precondition` eine Vorbedingung für die Ausführung eines `AbstractRefactorings`. Wird die Bedingung nicht erfüllt, so wird der ganze Refactoringablauf abgebrochen. Eine Definition der `Precondition` findet sich auch in theoretischen Ausarbeitungen von Thomas Ruhroth [RW07].

SubRefactoringCondition: Instanzen von `SubRefactoringCondition` geben eine Bedingung für die Ausführung eines Subrefactorings aus einem `ComplexRefactoring` an. Das ensprechende Subrefactoring wird immer wieder angestoßen, bis die `SubRefactoringCondition` nicht mehr erfüllt werden kann.

Die in einer `Condition` enthaltene `IContexQuerys` liefern im Gegensatz der Klasse `CalculationTemplate` (vergleiche Abbildung 6.37) Boolsche Werte bei Ausführung.

Klassen, die von `CalculationTemplate` erben (siehe Abbildung 6.37), beschreiben Abläufe eines Refactorings, in welchen Variablen unter Zuhilfenahme von `IContextQuery`-Objekten belegt werden können:

AfterTemplate: Die Klasse `AfterTemplate` entspricht der Definition von **After Templates** nach theoretischen Ausarbeitungen von Thomas Ruhroth [RW07]. Demnach ist es der Bestandteil eines Refactorings, in dem die eigentlichen Änderungen am Modell vorgenommen werden. Um mit Hilfe von OCL Änderungen am Modell vornehmen zu können, wurden im Ecore-Modell von UML/Z Methoden definiert und in den jeweiligen Klassen später implementiert (vergleiche dazu Angaben aus Kapitel 7). `AfterTemplate`-Instanzen können nur in `SimpleRefactoring`-Objekten enthalten sein. Damit sind Modifikationen eines Modells nur in einfachen Refactorings möglich.

BeforeTemplate: Die Klasse `BeforeTemplate` entspricht der Definition von **Before Templates** nach theoretischen Ausarbeitungen von Thomas Ruhroth [RW07]. In `BeforeTemplates` werden Teile von Modellelementen Variablen eines Refactorings zugewiesen. Ein `BeforeTemplate` wird vor der Validierung der `Precondition` angestoßen. In vielen Fällen werden durch Entscheidungen (`Decision`), die vom Benutzer getroffen werden, bereits im `RefactoringType` notwendige Variablen belegt, so dass ein `BeforeTemplate` leer bleiben kann.

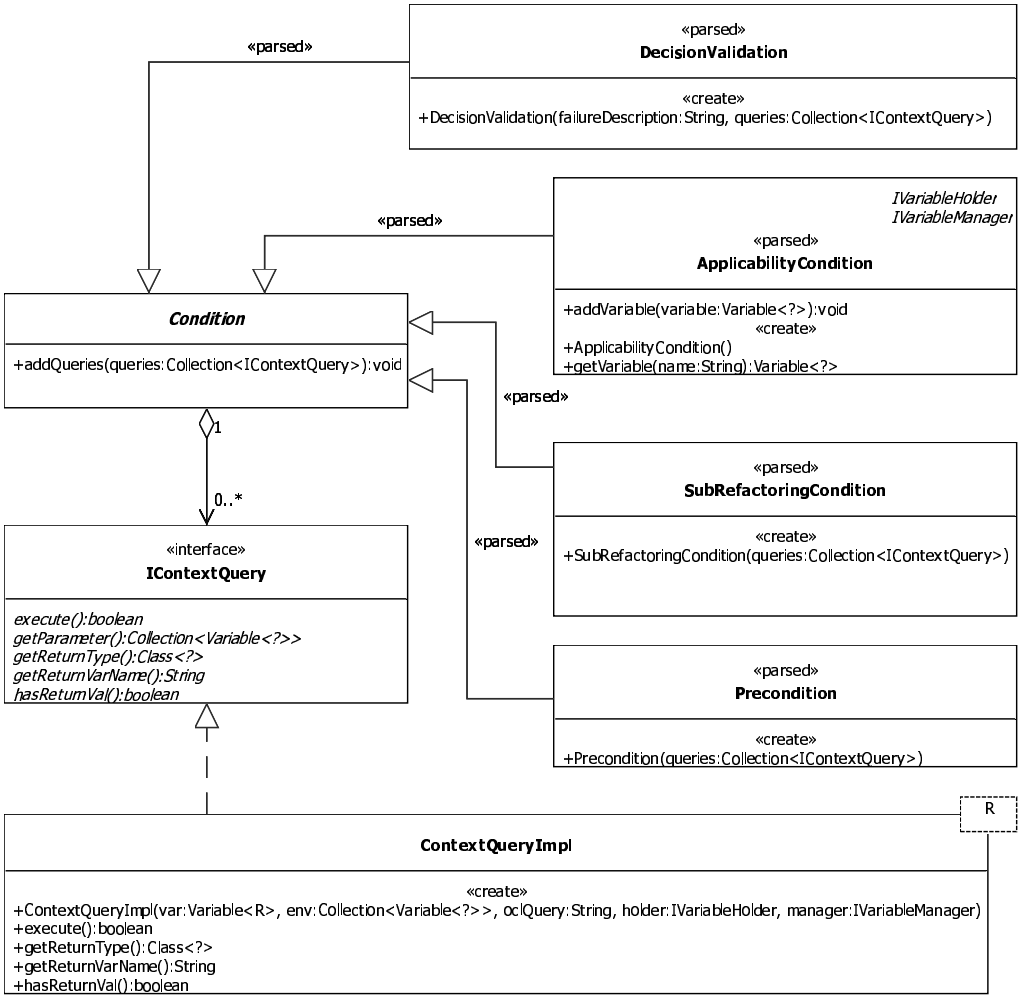


Abbildung 6.36: Klassendiagramm: von `Condition` abgeleitete Klassen

Calculation: Die Klasse `Calculation` entspricht der Definition von Calculations nach theoretischen Ausarbeitungen von Thomas Ruhroth [RW07]. Darin werden nicht nur Zuweisungen von Modellelementen, sondern im Wesentlichen Berechnungen auf Modellelementen und Mengen von Modellelementen durchgeführt. Dieser Bestandteil eines `AbstractRefactoring` unterscheidet sich strukturell nicht von einem `BeforeTemplate`, da beiden `IContextQuerys` zugeordnet sind, die nacheinander ausgeführt werden. In OCL-Anfragen der `Calculations` werden die Modellmengen und -elemente errechnet, die in einem weiteren Schritt in `AfterTemplates` bzw. `SubRefactorings` im zugrundeliegende Modell umgesetzt werden. Eine `Calculation` wird nach der Überprüfung der `Precondition` durchgeführt.

StumblingBlock: Eine Instanz der Klasse `StumblingBlock` ist der erste Bestandteil eines Refactorings, der beim Anstoßen des Rechenablaufs durchgeführt wird. Damit wird eine initiale Variablenmenge vor dem Ausführen eines Refactorings belegt. Auf den hier belegten Variablen werden Entscheidungen vom Benutzer getroffen (vergleiche Beschreibung zu `Decision` und Abbildung 6.38). Der `StumblingBlock` ist Bestandteil eines Refactoringtyps (`RefactoringType`) und wird damit im Ablauf eines angestoßenen Refactorings nur einmal durchlaufen. Dieser Teil eines `RefactoringType` ist vergleichbar mit einem `BeforeTemplate` in einem `AbstractRefactoring`.

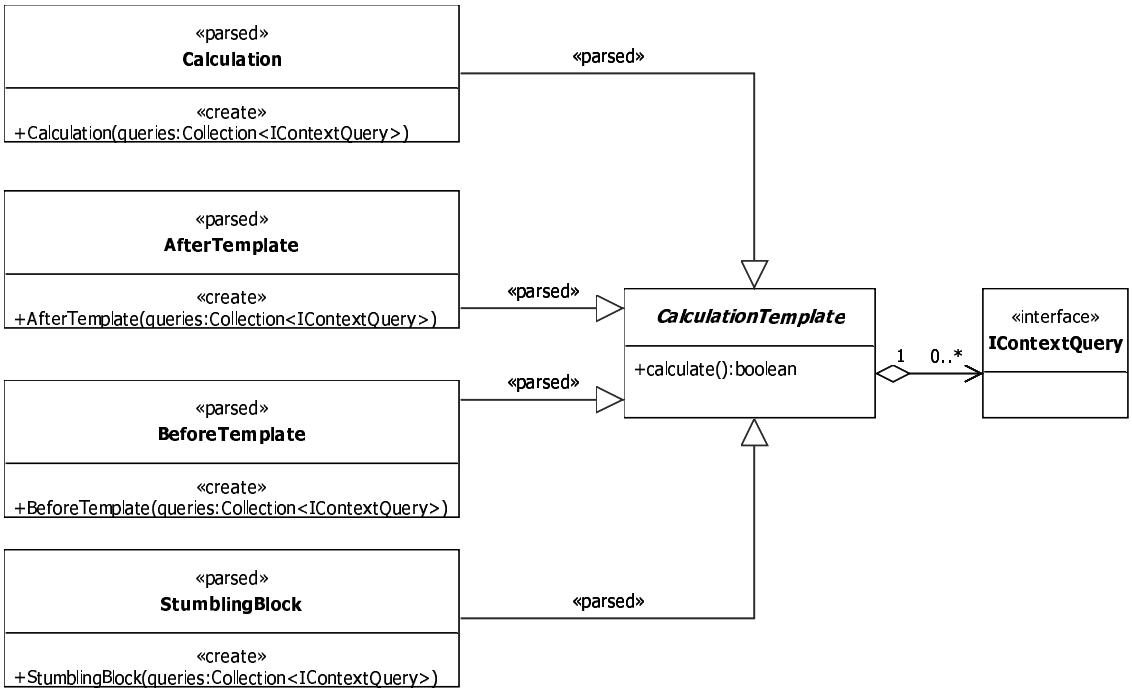


Abbildung 6.37: Klassendiagramm: von `CalculationTemplate` abgeleitete Klassen

`Decision`-Objekte entsprechen notwendigen Entscheidungen, die innerhalb der Durchführung eines Refactorings auftreten können und vom Anwender aufgelöst werden müssen. Die Datenstruktur, die einer `Decision` entspricht, ist der Abbildung 6.38 zu entnehmen.

Alle notwendigen Entscheidungen werden aus der Konfigurationsdatei eingelesen. Die Ent-

scheidungen sind in einem `RefactoringType` definiert. Einzelne `Decision`-Typen sind generisch ausgelegt, so dass z.B. eine ListQueryDecision auf beliebige Elementen ausführbar ist.

Im Folgenden sind die drei `Decision`-Typen näher aufgeschlüsselt.

SplitPredicateDecision: Erfragt eine Teilung eines Prädikats und überprüft die Gültigkeit der Aufteilung.

ListQueryDecision: Bietet dem Anwender eine Auswahl von Elementen an und definiert erlaubte Selektionen.

StringInputDecision: Erlaubt eine Texteingabe und prüft diese auf Gültigkeit.

Variablen, welche unter anderem Rückgabewerte von OCL-Anfragen enthalten können, sind ein wesentlicher Bestandteil der Refactorings. Name und Typ einer Variable werden aus einer Konfigurationsdatei eingelesen. Die Verwaltung der Variablen übernimmt ein `IVariableHolder` bzw. ein `IVariableManager`: über den `IVariableHolder` lassen sich `Variable`-Objekte erfragen, über den `IVariableManager` können `Variable`-Objekte dem verwaltenden Objekte hinzugefügt werden. Jede `ContextQueryImpl` hat genau einen `IVariableHolder` und kann auch eine Referenz auf maximal einen `IVariableManager` enthalten. Wie der Abbildung 6.39 zu entnehmen ist, werden die Interfaces `IVarialbeHolder` und `IVariableManager` von der Klasse `AbstractRefactoring` implementiert. In der Klasse `VariableToElementMap` werden Instanzen der Variablen in `AbstractRefactoring`s vorgehalten.

Außer durch eine `ContextQueryImpl` kann eine Variable auch über eine Instanz von `VariableAssignment` aus einem `AbstractRefactoring` (fungiert als `IVariableHolder`) in ein anderes `AbstractRefactoring` (fungiert als `IVariableManager`) übertragen werden.

Um den Fortschritt eines Refactorings bei der Ausführung dem Anwender anzeigen und Eingaben des Anwenders engegennehmen zu können, ist die GUI für Refactorings, wie in Abbildung 6.40 dargestellt, entworfen worden. Dabei existiert zu jeder `Decision` (vergleiche Abbildung 6.38) eine `RefactoringDecisionPage`, die die Entscheidung dem Anwender anzeigt. Ein `RefactoringWizard` besteht dabei nicht nur aus einer `RefactoringStartPage`, sondern aus einer Menge von `WizardPage`s gemäß der Superklasse `Wizard`.

Fortschrittsereignisse werden über die Interfaces gemäß Abbildung 6.41 an die `IProgressListener` weitergereicht werden. Dabei können aus einem `IProgressEndEvent` neben einer Beschreibung (`getDescription`) und einem Fortschrittswert (`getProgressValue`) Erfolgsangaben abgefragt werden.

6.3.5.2 Verhalten

Folgende drei Sequenzdiagramme (Abbildungen 6.42, 6.43 und 6.44) stellen die wichtigsten Abläufe beim Finden und Durchführen von Refactorings vor.

Abbildung 6.42 beschreibt den Ablauf beim Aufruf der Methode `getRefactoringsFor` der Klasse `RefactoringManager`. Hierbei werden zwei Fälle unterschieden:

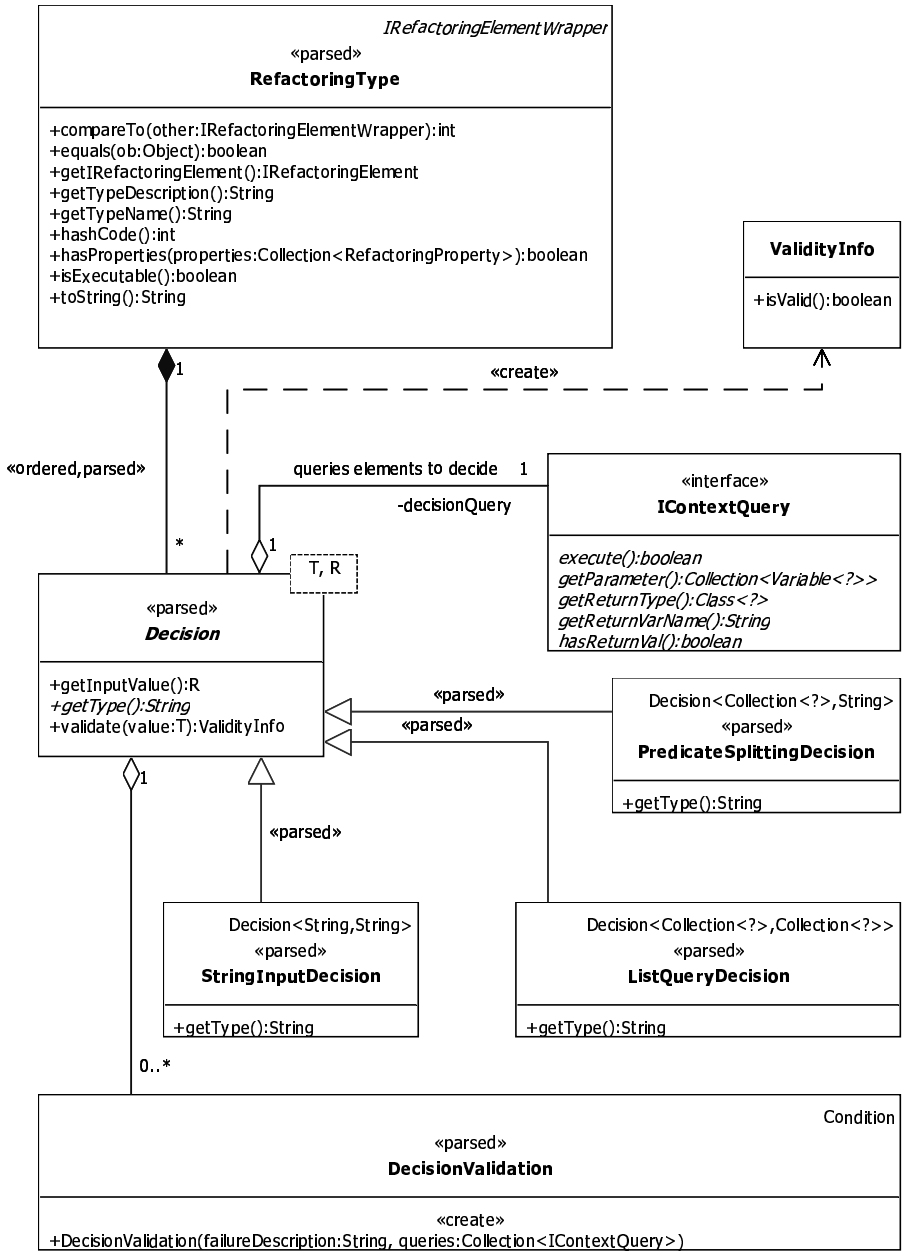


Abbildung 6.38: Klassendiagramm: Decisions

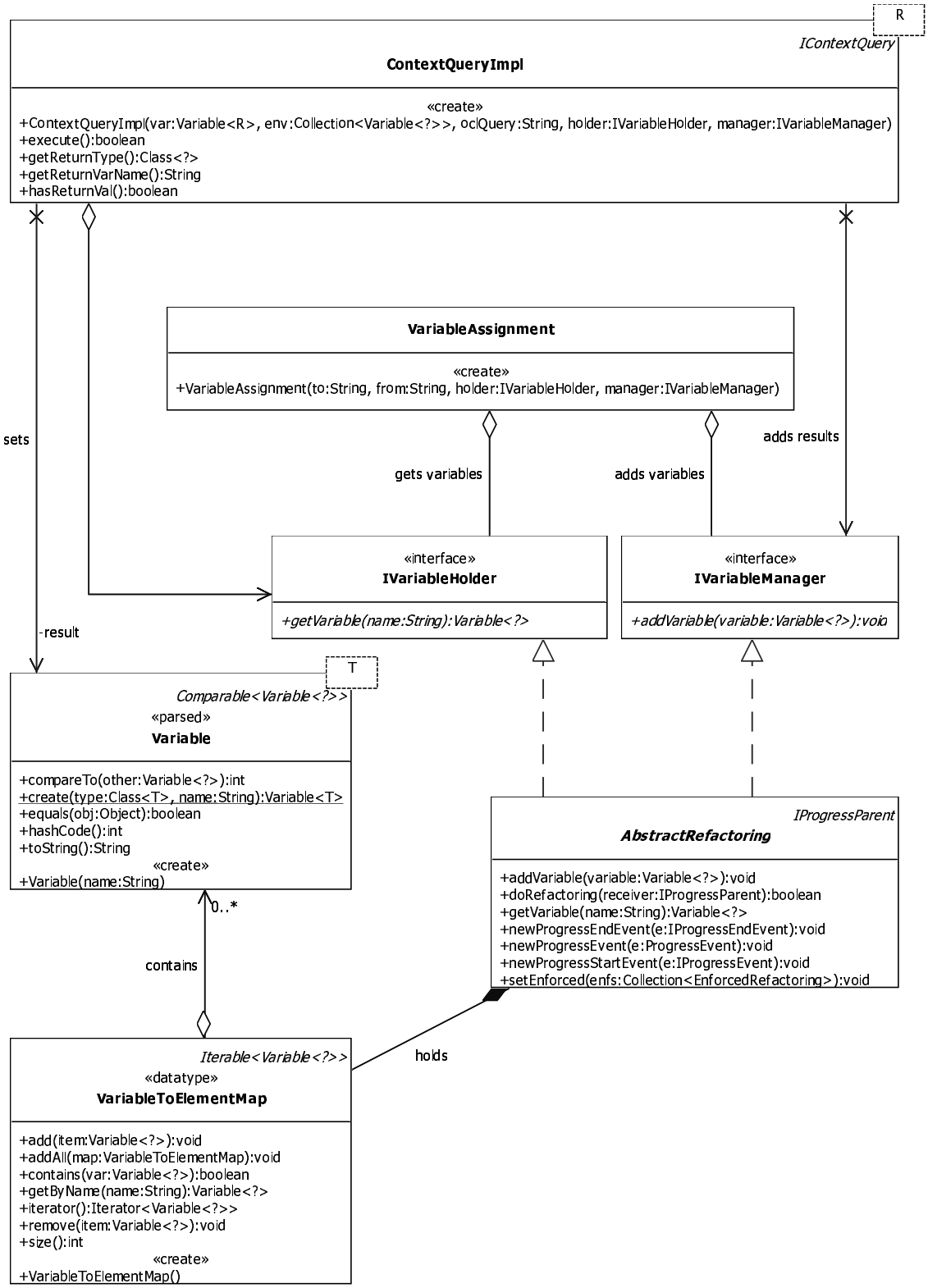


Abbildung 6.39: Klassendiagramm: Variablenmanagement in Refactorings

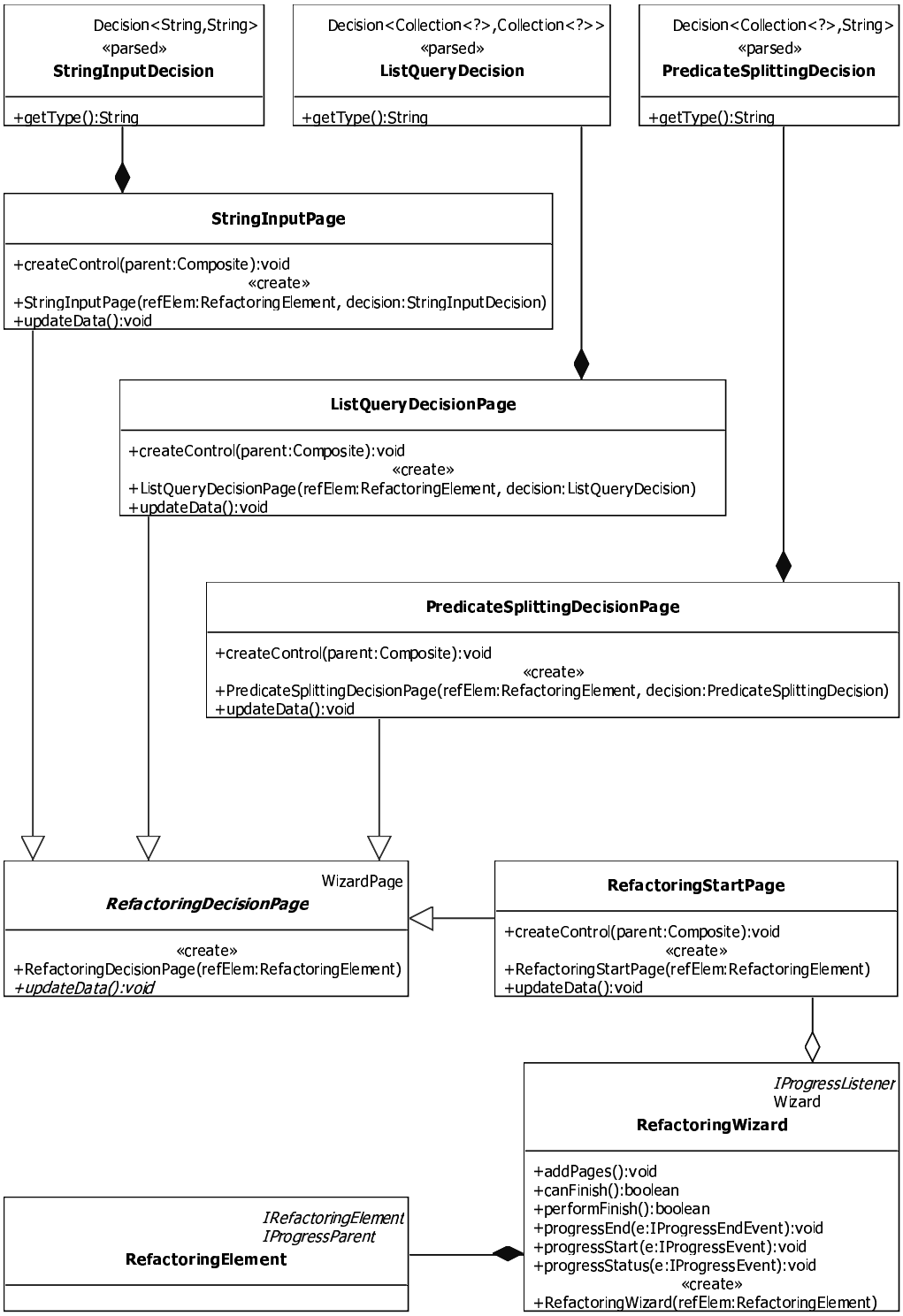


Abbildung 6.40: Klassendiagramm: Klassen der Refactoring-GUI

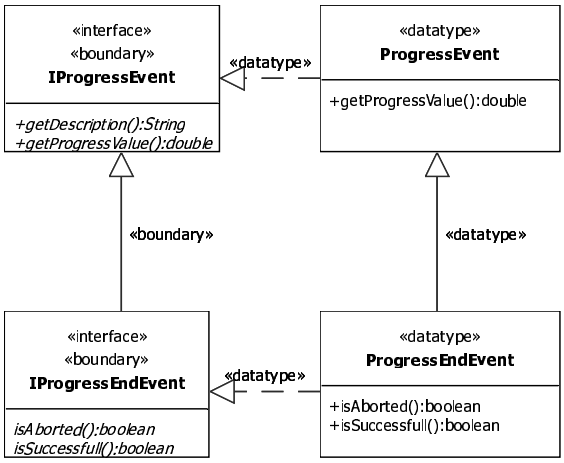


Abbildung 6.41: Klassendiagramm: Fortschrittsangaben bei Ausführung von Refactorings

1. Die `ApplicabilityConditions` eines Refactoringtypen ist erfolgreich validiert worden. In diesem Fall wird ein `IRefactoringElement`-Objekt erzeugt und der Menge zurückzuliefernder `IRefactoringElementWrapper` beigefügt.
2. Es ist ein Fehler bei der Überprüfung der `ApplicabilityConditions` aufgetreten. Dabei wird der Refactoringtyp selbst (`RefactoringType`) der Menge der `IRefactoringElementWrapper` hinzugefügt.

Abbildung 6.35 entsprechend existieren zwei Arten von Refactorings: `SimpleRefactoring` und `ComplexRefactoring`. Beide erben von der Klasse `AbstractRefactoring`, die die gemeinsamen Funktionalitäten der beiden Refactoringarten abbildet. Das Sequenzdiagramm der Abbildung 6.43 zeigt den gemeinsamen Ablauf.

Falls die `Precondition` und die `Calculation` in der Operation `evaluatePrecondition-AndCalculation` nicht **true** liefern, so wird die Menge der zu erzwingenden Refactorings (`EnforcedRefactoring`) vollständig durchgeführt. Damit wird beabsichtigt, die `Precondition` (und damit auch die `Calculation`) abschließend erfolgreich durchführen und zu Abläufen, die von der Refactoringart abhängig sind, vordringen zu können.

Falls das `AbstractRefactoring` ein `SimpleRefactoring` ist, so wird nach der erfolgreichen Validierung der `Precondition` und der anschließenden Durchführung der `Calculation`, das `AfterTemplate` abgearbeitet. Hierbei werden Änderungen im Modell realisiert.

Ist das `AbstractRefactoring` ein `ComplexRefactoring`, so entspricht der anschließende Ablauf der Abbildung 6.44. Hierbei werden alle `SubRefactoring`-Objekte durchgeführt, solange die `SubRefactoringCondition` **true** liefert.

6.3.6 OCLWrapper

Um OCL-Anfragen sowie Änderungsanweisungen an UML/Z-Modelle zu ermöglichen, wurde die OCL-Bibliothek[5] von eclipse.org eingebunden. Die Komponente wird nicht nur zur

[5] http://www.eclipse.org/modeling/mdt/?project=ocl

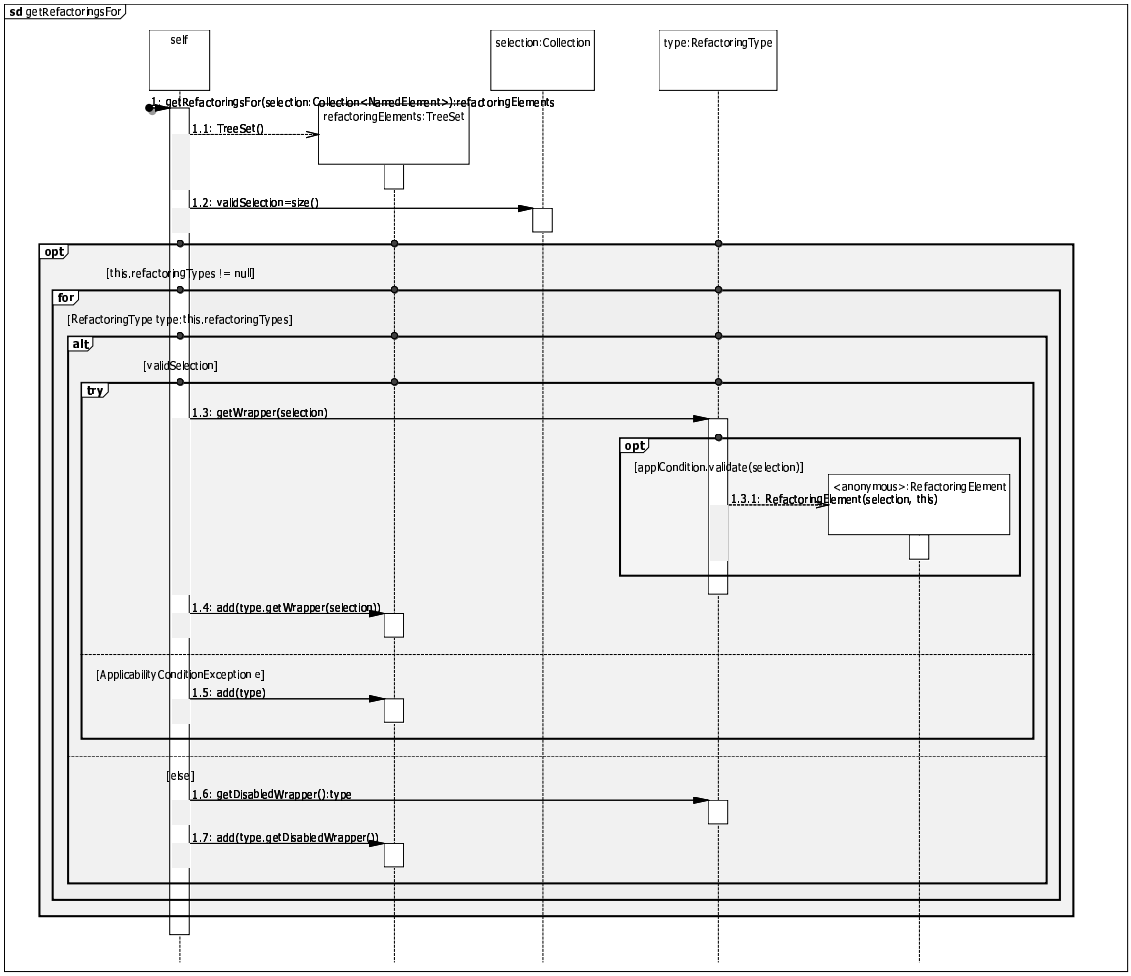


Abbildung 6.42: Sequenzdiagramm: Finden von Refactorings zu Modellelementen

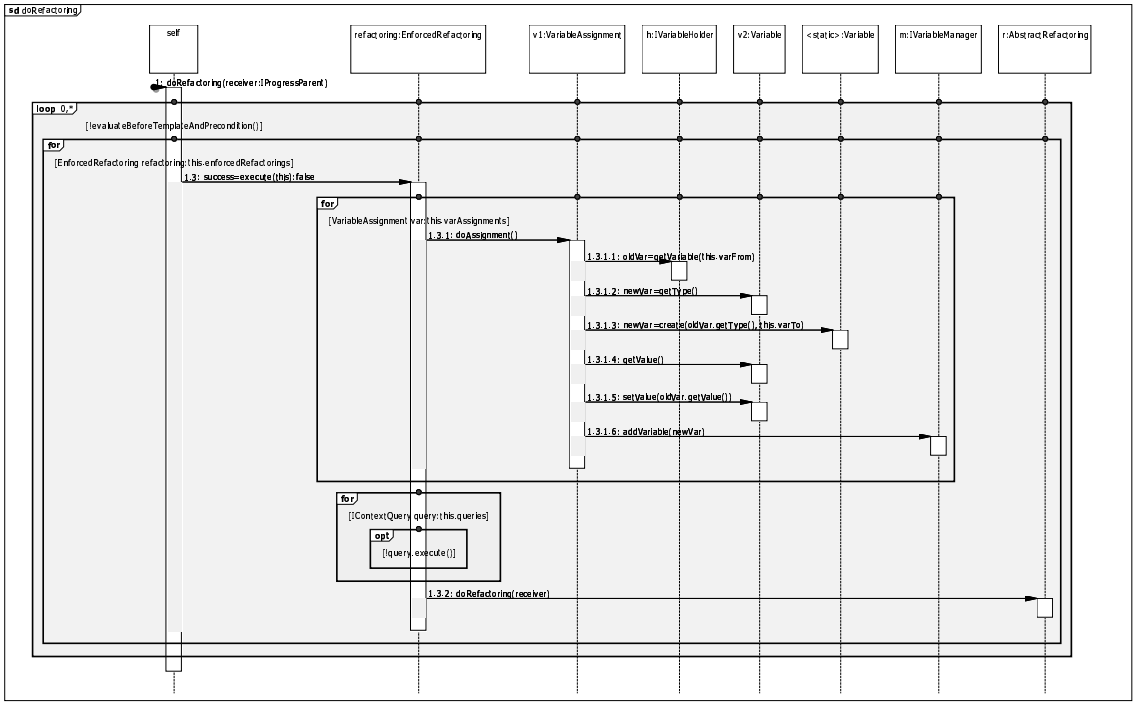


Abbildung 6.43: Sequenzdiagramm: Ausführen eines Refactorings

Definition von Messungen eines MQP, sondern auch zur Konfiguration von Refactorings verwendet. Es werden Klassen zum Parsen und für die Auswertung von OCL-Ausdrücken zur Verfügung gestellt. Für Komponenten, die OCL-Ausdrücke auf UML/Z-Modellen verwenden, werden die benötigten Methoden und Funktionen zur Erzeugung und Auswertung von OCL-Ausdrücken sowie Methoden zur Erzeugung von Variablen in der OCL-Umgebung `OCL` bereitgestellt.

Die Zusammenhänge der **OCLWrapper**-Klassen sind in Abbildung 6.45 veranschaulicht. Mit Hilfe der Klasse `OCLParser` wird zunächst eine Umgebung vom Typ `OCLParseEnvironment` erzeugt.

Die Klasse `OCLParseEnvironment` stellt mit den Operationen `defineOperations`, `parse`, `parseContextFree` sowie `parseAndEvaluate` die zum Parsen und Auswerten von OCL-Ausdrücken notwendigen Funktionalitäten bereit. Geparste OCL-Ausdrücke werden dabei in der Klasse `OCLQuery` bzw. `OCLContextFreeQuery`gekapselt.

Berücksichtigt wurden hierbei sowohl kontextfreie als auch kontextbezogene OCL-Ausdrücke gemäß [OMG06]. Kontextfreie Ausdrücke werden mit Hilfe der Klasse `OCLHelper`, kontextbezogene Ausdrücke mit der Klasse `OCL` erzeugt.

Ein Kontext ist in diesem Zusammenhang eine Beschreibung von Elementen bzw. Elementtypen, auf die sich die OCL-Ausdrücke beziehen. Das Charakteristikum kontextfreier Ausdrücke ist, dass der Kontext nicht, wie bei OCL üblich, textuell angegeben werden muss und

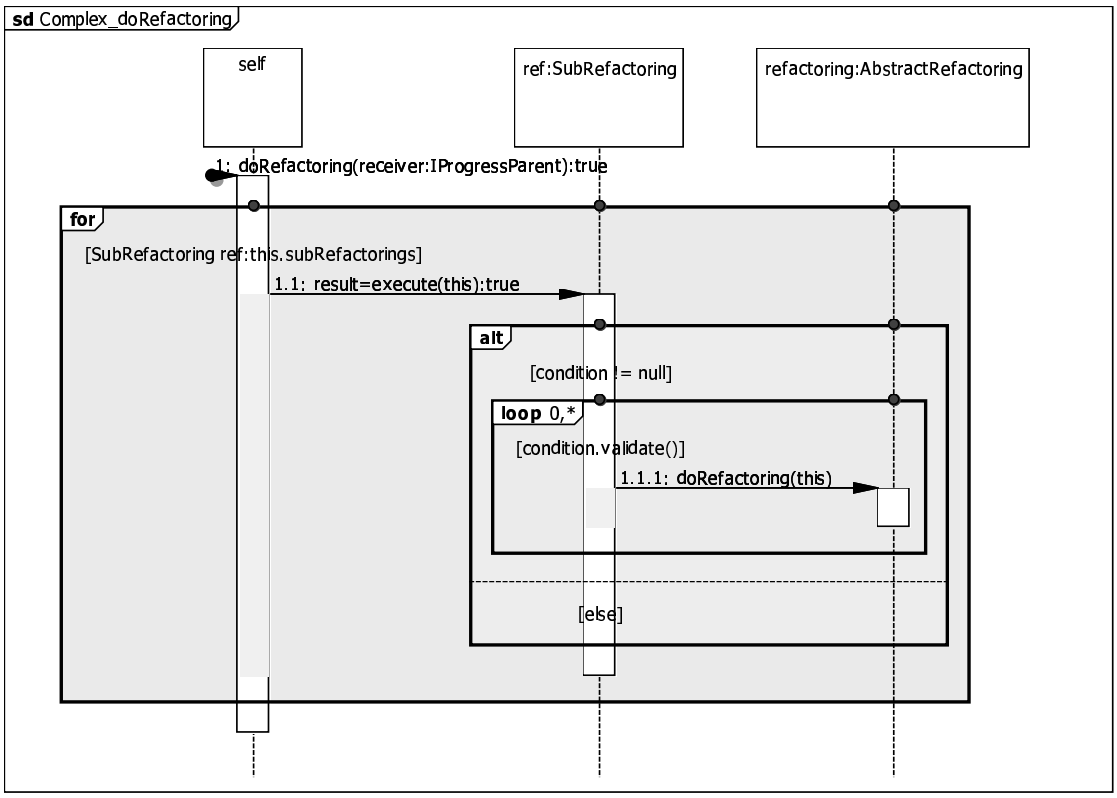


Abbildung 6.44: Sequenzdiagramm: Ausführen eines komplexen Refactorings

somit unterschiedliche Ausprägungen annehmen kann. Vielmehr werden zur Laufzeit Modellelemente (`EClassifier`) als sogenannte Klassifizierer übergeben. Dies ist z.B. hilfreich, um für alle Klassifizierer, die ein Attribut **name** haben, einen OCL-Ausdruck auswerten zu können, ohne diesen speziell für jeden einzelnen Klassifizierer separat anzulegen.

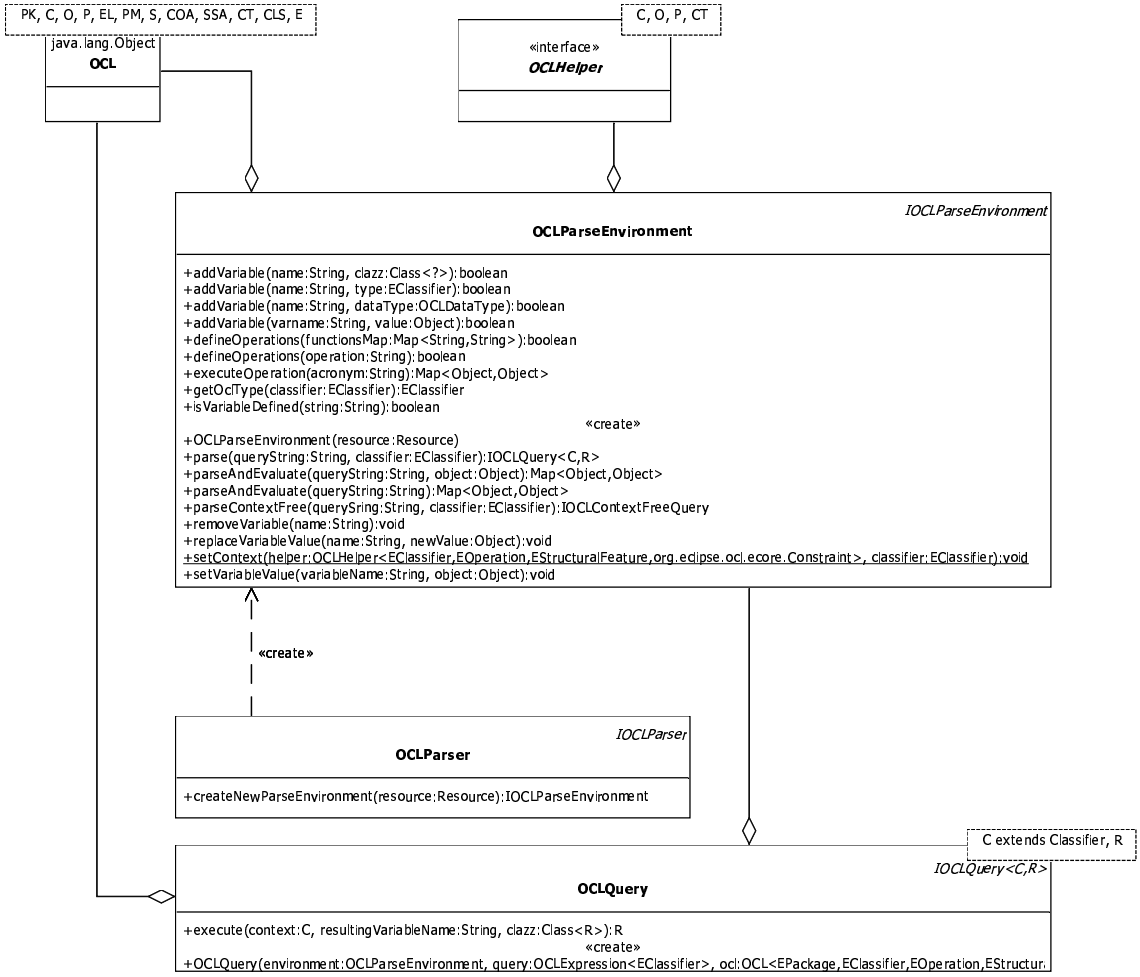


Abbildung 6.45: Klassendiagramm: OCLWrapper-Komponente

6.3.7 CZTWrapper

Um Z-Ausdrücke für Prädikate in Vor- und Nachbedingungen von Operationen als auch zur Beschreibung von Typen nutzen zu können, wurde die CZT-Bibliothek eingebunden [MU05]. Diese stellt Klassen zum Parsen und Durchlaufen von Z-Ausdrücken zur Verfügung.

Für Komponenten, die auf Z-Prädikaten und Z-Ausdrücken arbeiten, werden CZT-interne Datenstrukturen, die als Abstract Syntax Tree (AST) vorliegen, wie `Pred` oder allgemein `Term` (vergleiche CZT-Javadoc[6]) in Wrapper-Objekte verpackt.

[6] http://czt.sourceforge.net/

Die Zusammenhänge der CZTWrapper-Klassen sind dem Klassendiagramm der Abbildung 6.46 zu entnehmen. Dabei werden von der Klasse `CZTUtil` Funktionalitäten zur Verfügung gestellt, die das Parsen von Zeichenketten erlauben.

Die Klassen `ZNameCollector` und `ZNameExchanger` implementieren beide das Interface `ZNameVisitor`, das von den CZ-Tools bereitgestellt wird. Beide Klassen werden von `CZTUtil` genutzt, um geparste Datenstrukturen zu durchlaufen, Variablennamen zu finden und entsprechend Klassenfunktion zu verarbeiten:

- `ZNameExchanger`: Die Klasse durchläuft den AST, um gegebene Variablennamen durch entsprechende Objekte der Klasse `NamedElementWrapper` zu ersetzen. In einem Objekt der Klasse `NamedElementWrapper` werden Modellelemente des dazugehörigen UML/Z-Modells verpackt. Bei der Änderung des Namens dieser Elemente, ändert sich dieser im verwendenden Z-Ausdruck ebenfalls.
- `ZNameCollector`: Es können alle verwendeten Elemente des dazugehörigen UML/Z-Diagramms gesammelt und als `Collection` angefordert werden.

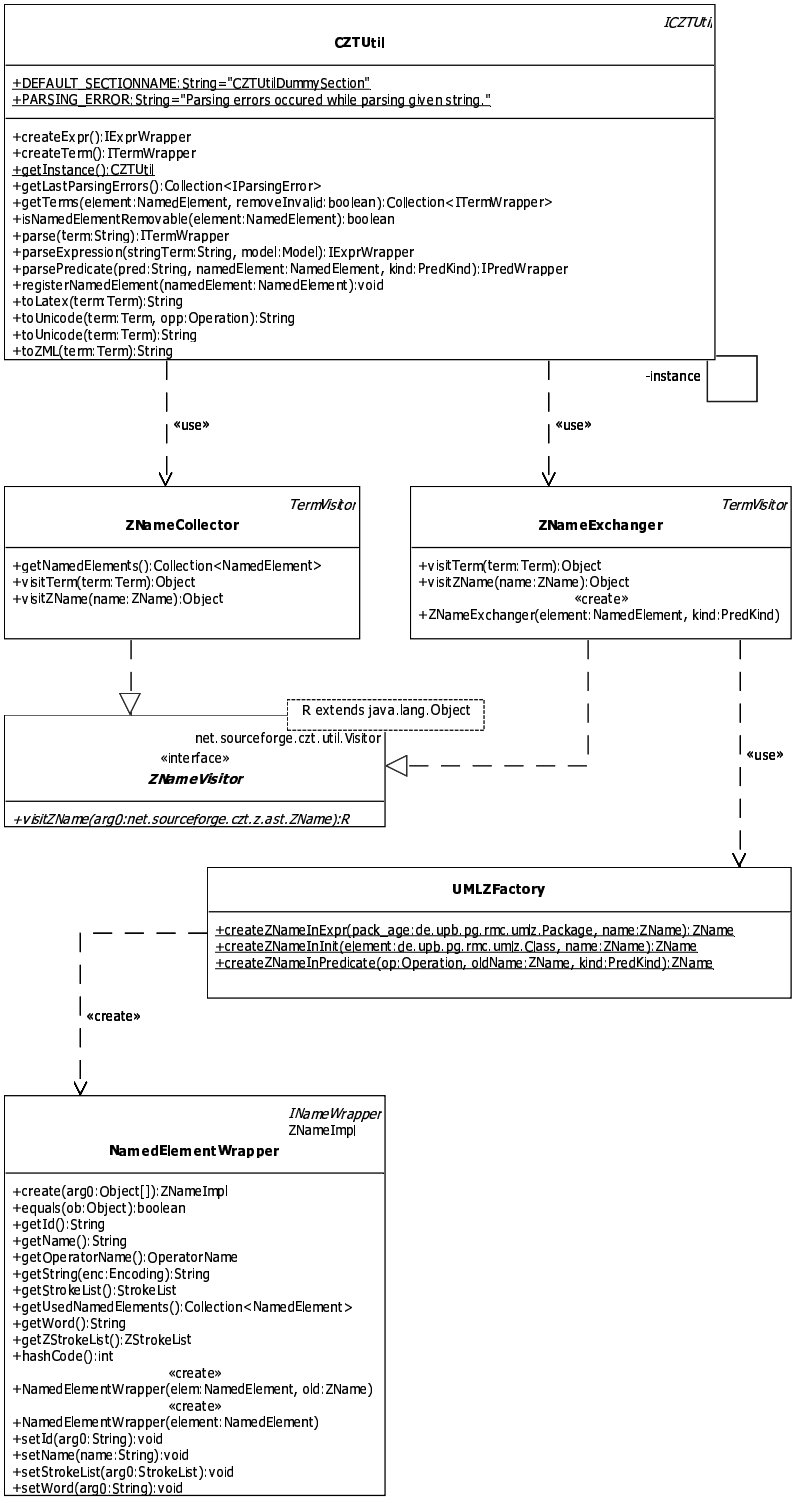


Abbildung 6.46: Klassendiagramm: CZTWrapper-Komponente

7 Entwicklerdokumentation

„Das Meta-Huhn.“

Peter, auf die Frage: Was war zuerst da, Henne oder Ei, und der Nachfrage: Wer hat denn das erste Ei gelegt.

7.1 Installation der Entwicklungsumgebung

7.1.1 Voraussetzungen

Zur Weiterentwicklung von RMC benötigen Sie folgende Software:

- Java SDK[1] ab Version 1.6
- „Eclipse[2] for RCP/Plug-in Developers“ ab Version 3.3.2
- RMC-Quellcode[3]
- Eine funktionierende Internetverbindung während der Installation

7.1.2 Installation von Eclipse und der Plugins

Um RMC kompilieren zu können, benötigen Sie den RMC-Quellcode und Eclipse mit einigen zusätzlichen Plugins. Für die Installation dieser Plugins und die Einrichtung Ihrer Arbeitsumgebung gehen Sie folgendermaßen vor:

1. Installieren Sie das Java SDK[4] ab Version 1.6 und „Eclipse[5] for RCP/Plug-in Developers“ ab Version 3.3.2.
2. Starten Sie Eclipse. Sie sehen nun den Dialog **Select a Workspace**.
3. Wählen Sie als Workspace ein leeres Verzeichnis aus, und bestätigen Sie Ihre Auswahl mit [OK]. Sie sehen nun die Eclipse **Welcome**-Seite.

[1] http://java.sun.com
[2] http://www.eclipse.org/downloads
[3] http://www.cs.uni-paderborn.de/fachgebiete/fg-engels/lehre/ws0708/pg-rmc-teil2/download.html
[4] http://java.sun.com
[5] http://www.eclipse.org/downloads

4. Schließen Sie die **Welcome**-Seite. Sie sehen nun eine leere Eclipse Arbeitsumgebung.
5. Klicken Sie im Menü auf ⟨`Help`⟩→⟨`Software Updates`⟩→⟨`Find and Install`⟩. Sie sehen nun den Dialog **Feature Updates**.
6. Wählen Sie die Option ⟨`Search for New Features to Install`⟩ und bestätigen Sie Ihre Auswahl mit [Next]. Sie sehen nun den Dialog **Update Sites to Visit**.
7. Markieren Sie die Update Site **Europa Discovery Site** und bestätigen Sie Ihre Auswahl mit [Finish]. Es erscheint der Dialog **Update Manager**, und einige Sekunden später der Dialog **Update Site Mirrors**.
8. Bestätigen Sie die Auswahl eines Mirrors mit [OK]. Es erscheint wieder der Dialog **Update Manager**.
9. Eclipse benötigt nun einige Sekunden zum Ermitteln der verfügbaren Updates. Warten Sie, bis dieser Vorgang abgeschlossen ist und der Dialog **Search Results** erscheint.
10. Markieren Sie nun folgende Plugins aus der Liste **Enabling Features**, wie in Abbildung 7.1 zu sehen:
    - EMF Model Query 1.1
    - EMF Model Query OCL Integration 1.1
    - EMF Validation Framework 1.1
    - EMF Model Transaction 1.1
    - EMF Model Transaction Workbench Integration 1.1
    - Object Constraint Language (OCL) 2.0 1.1
    - OCL 2.0 Binding For UML 1.1
    - Object Constraint Language (OCL) 2.0 Compatibility 1.1

    Markieren Sie außerdem folgende Plugins aus der Liste **Models and Model Development**:
    - Eclipse Modeling Framework (EMF) Runtime + End User Tools
    - Graphical Modeling Framework (Europa Edition) 2.0
    - UML 2 End User Features 2.1

    Nachdem Sie alle Plugins markiert haben bestätigen Sie Ihre Auswahl mit [Next]. Sie sehen nun den Dialog **Feature License**.
11. Bestätigen Sie die Lizenzen der zu installierenden Plugins durch Auswahl der Option ⟨`I accept the terms in the license agreements`⟩ und klick auf [Next]. Sie sehen nun den Dialog **Installation**.
12. Bestätigen Sie die Installation der Plugins mit [Finish]. Es erscheint der Dialog **Update Manager**.
13. Eclipse lädt nun die Plugins herunter. Warten Sie, bis dieser Vorgang abgeschlossen ist. Sie sehen nun den Dialog **Feature Verification**.
14. Bestätigen Sie erneut die Installation der Plugins mit [Install All]. Es erscheint erneut der Dialog **Update Manager**.
15. Eclipse installiert nun die Plugins. Warten Sie, bis dieser Vorgang abgeschlossen ist. Sie sehen nun den Dialog **Install/Update**.

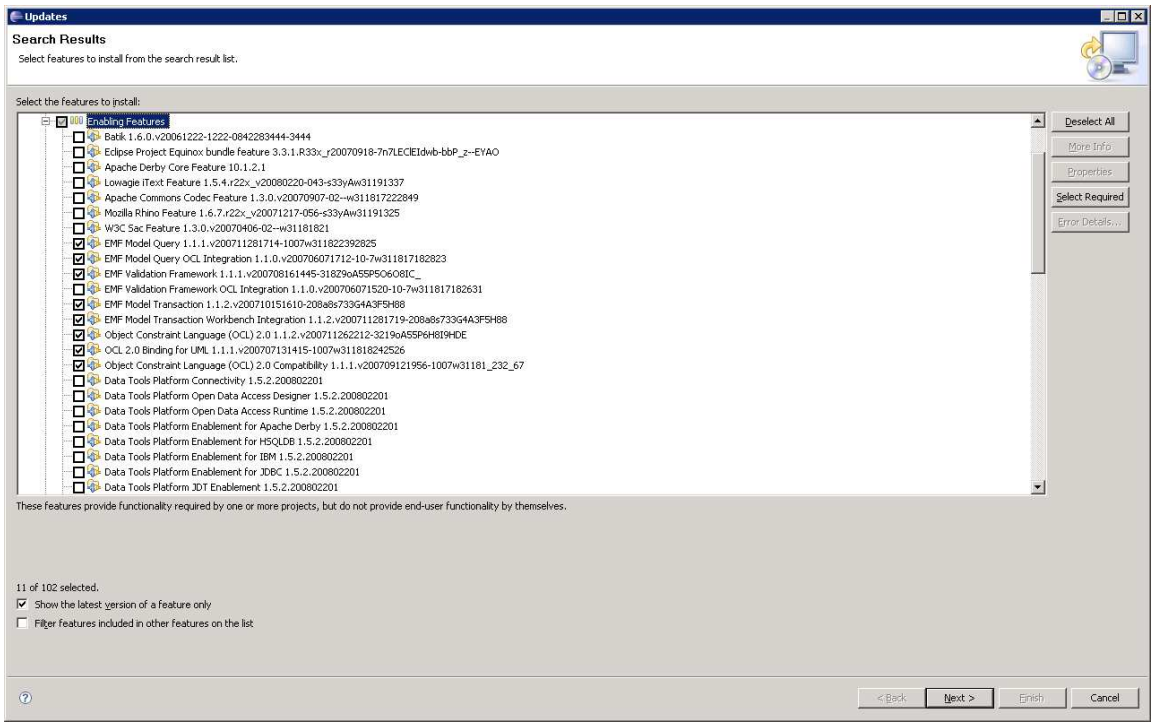

Updates
Search Results
Select features to install from the search result list.
Select the features to install:
Enabling Features
Batik 1.6.0.v20061222-1222-0842283444-3444
Eclipse Project Equinox bundle feature 3.3.1.R33x_r20070918-7n7LECIEIdwb-bbP_z--EYAO
Apache Derby Core Feature 10.1.2.1
Lowagie iText Feature 1.5.4.r22x_v20080220-043-s33yAw31191337
Apache Commons Codec Feature 1.3.0.v20070907-02--w311817222849
Mozilla Rhino Feature 1.6.7.r22x_v20071217-056-s33yAw31191325
W3C Sac Feature 1.3.0.v20070406-02--w31181821
EMF Model Query 1.1.1.v200711281714-1007w311822392825
EMF Model Query OCL Integration 1.1.0.v200706071712-10-7w311817182823
EMF Validation Framework 1.1.1.v200708161445-318Z9oA55P5O6O8IC_
EMF Validation Framework OCL Integration 1.1.0.v200706071520-10-7w311817182631
EMF Model Transaction 1.1.2.v200710151610-208a8s733G4A3F5H88
EMF Model Transaction Workbench Integration 1.1.2.v200711281719-208a8s733G4A3F5H88
Object Constraint Language (OCL) 2.0 1.1.2.v200711262212-3219oA55P6H8I9HDE
OCL 2.0 Binding for UML 1.1.1.v200707131415-1007w311818242526
Object Constraint Language (OCL) 2.0 Compatibility 1.1.1.v200709121956-1007w31181_232_67
Data Tools Platform Connectivity 1.5.2.200802201
Data Tools Platform Open Data Access Designer 1.5.2.200802201
Data Tools Platform Open Data Access Runtime 1.5.2.200802201
Data Tools Platform Enablement for Apache Derby 1.5.2.200802201
Data Tools Platform Enablement for H5QLDB 1.5.2.200802201
Data Tools Platform Enablement for IBM 1.5.2.200802201
Data Tools Platform Enablement for JDBC 1.5.2.200802201
Data Tools Platform JDT Enablement 1.5.2.200802201
Deselect All
More Info
Properties
Select Required
Error Details...
These features provide functionality required by one or more projects, but do not provide end-user functionality by themselves.
11 of 102 selected.
Show the latest version of a feature only
Filter features included in other features on the list
< Back
Next >
Finish
Cancel

Abbildung 7.1: Installation: Plugins aus Liste **Enabling Features**

16. Bestätigen Sie mit [Yes], dass Eclipse neu gestartet werden soll. Eclipse startet nun neu. Nach dem Neustart erscheint der Dialog **Select a Workspace**.
17. Als Workspace sollte der von Ihnen unter 3. erstellte Workspace eingetragen sein. Bestätigen Sie diese Auswahl mit [OK]. Sie sehen nun erneut eine leere Eclipse Arbeitsumgebung.

7.1.3 Importieren des RMC-Quellcodes

Sie haben nun Eclipse und alle für die Weiterentwicklung von RMC benötigten Plugins installiert und einen leeren Workspace erstellt. Als nächstes müssen Sie den RMC Quellcode in diesen Workspace importieren.

1. Entpacken Sie den RMC Quellcode in ein leeres Verzeichnis.
2. Klicken Sie in Eclipse auf das Menü ⟨`File`⟩→⟨`Import`⟩. Sie sehen nun den Dialog **Select**.
3. Markieren Sie den Eintrag ⟨`General`⟩→⟨`Existing Projects into Workspace`⟩, und bestätigen Sie Ihre Auswahl mit [Next]. Sie sehen nun den Dialog **Import Projects**.
4. Klicken Sie auf den Button [Browse]. Sie sehen nun den Dialog **Select root directory**.

5. Wählen Sie das Verzeichnis aus, in das Sie den RMC Quellcode entpackt haben, und bestätigen Sie Ihre Auswahl mit [OK]. Sie sehen nun erneut den Dialog **Import Projects**, und in der **Projects**-Liste sollten mehrere Projekte aufgelistet und markiert sein.
6. Bestätigen Sie die Auswahl mit [Finish]. Eclipse importiert nun alle Projekte in Ihren Workspace.
7. Sie haben nun alle Projekte importiert, Eclipse kompiliert diese automatisch. Während Eclipse alle Projekte kompiliert, erscheint der Hinweis **Building Workspace** unten rechts im Eclipse-Fenster in der Statusleiste. Warten Sie, bis dieser Vorgang abgeschlossen ist. Ihre Arbeitsumgebung ist jetzt für die Weiterentwicklung von RMC vorbereitet.

7.1.4 Erstellen von RMC über das Ant-Script

Neben dem Kompilieren von RMC in Eclipse können Sie RMC auch über ein Ant-Script kompilieren. Dieses finden Sie im RMC Quellcode Archiv unter dem Namen src
build.xml. Das Ant-Script kann z.B. für die automatisierte Erstellung von RMC durch einen Buildserver verwendet werden, für die meisten Anwendungsfälle sollten sie RMC aber ohne Ant-Script direkt in Eclipse bearbeiten können. Deshalb wird die Verwendung und Konfiguration des Ant-Scripts hier auch nur grob beschrieben.

1. Das Build-Script benötigt einen SVN-Client, unter Windows z.B. TortoiseSVN[6]. Installieren Sie einen SVN-Client.
2. Das Build-Script benötigt ein Eclipse, welches wie oben beschrieben installiert wurde. Zusätzlich muss in diesem Eclipse das Plugin Subclipse[7] installiert werden.
3. Sie müssen nun das Build-Script konfigurieren. Dafür müssen Sie zunächst die Datei src
   my.properties anpassen. Hinweise zu den einzelnen Konfigurationsmöglichkeiten finden Sie als Kommentare in der Datei.
4. Als nächstes müssen Sie abhängig von ihrem Betriebssystem entweder die Datei src
   setenv.cmd (Windows) oder src
   setenv.sh (Linux) erstellen und anpassen. Vorlagen für diese Dateien finden Sie unter src
   setenv.cmd.sample bzw. src
   setenv.sh.sample.
5. Sie können nun das Build-Script starten. Öffnen Sie dafür eine Konsole und wechseln Sie in das src Verzeichnis.
6. Unter Windows rufen Sie nun zunächst setenv.cmd auf, anschließend ant, z.B. mit **ant jar**.
7. Unter Linux rufen Sie das Build-Script z.B. mit **setenv.sh 'ant jar'** auf.

[6] http://tortoisesvn.tigris.org
[7] http://subclipse.tigris.org

7.1.5 Voraussetzungen

- Ein Subversion Client, z.B. Tortoise-SVN
- Ein Verzeichnis mit einer Eclipse-Installation, die wie in der Installationsanleitung für die Entwicklungsumgebung beschrieben installiert wurde. Wichtig ist, dass auch das Subclipse-Plugin installiert wurde.
- Grundkenntnisse über die Verwendung von Apache Ant oder genügend Zeit und die Bereitschaft, sich diese anzueignen.

RMC kann sowohl in Eclipse als auch über ein Build-Script erstellt werden. Das Build-Script kann z.B. von einem Build-Server verwendet werden, um automatisiert regelmäßig Builds zu erstellen. Das Build-Script kompiliert alle Projekte und erstellt .jar Dateien mit den kompilierten RMC Eclipse-Plugins, es erstellt JavaDoc-Dokumentation zu den Projekten, und es kann ein Setup-Archiv erzeugen. Das Build-Script besteht aus mehreren Ant Dateien. Die Hauptdatei befindet sich im RMC-Repository unter `/src/build.xml`.

7.1.6 Build-Tasks

Das Ant-Script enthält folgende Ant-Targets:

compile: Dieser Task ruft das clean-Target auf, kompiliert alle Dateien im `${srcBase}` Ordner und kopiert die erzeugten .class Dateien nach ${destBase}.

jar: Wie compile, zusätzlich werden für alle Plugins .jar-Archive erstellt und nach `${distBase}` kopiert.

install: Beginnt mit dem jar-Target. Die Plugins werden anschließend ins Eclipse Plugin-Verzeichnis `${eclipseBase}\Plugins` kopiert.

installwin: Kopiert eine leere Eclipse Version (für Windows) aus `${envBase}\Windows \eclipsebase` nach `${envBase}\windows \testbuild-\${svn.Revision}`, und installiert dort die RMC Plugins.

createsetup: Erzeugt ein .zip-Archiv mit allen vom jar-Target erstellten RMC Plugins.

postsetup: Dieser Task wird vom Build-Server benötigt.

clean: Löscht die Verzeichnisse `${destBase}` und `${distBase}`.

javadoc: Erzeugt die Javadoc-Dokumentation im Verzeichnis `${docBase}\all`

model-javadoc: Erzeugt die Javadoc-Dokumentation nur für das Modell.

checkout: Checkout des aktuellen Repository-Quellcodes nach `${srcBase}`.

7.1.7 Konfiguration des Build-Scripts

Das Build-Script wird über mehrere Dateien konfiguriert: `my.properties`, `setenv.cmd` (Windows) und `setenv.sh` (Linux). Bei allen Dateien sollten als Verzeichnistrenner die jeweils gleichen Zeichen wie in den Beispielen (\ bzw. /) verwendet werden.

my.properties: Hier werden verschiedene Verzeichnisse konfiguriert. Die Einstellungen sind direkt in der Datei beschrieben. Das RMC Repository enthält eine Beispielkonfiguration unter `/src/my.properties.sample`

setenv.cmd: Hier werden verschiedene Verzeichnisse und Umgebungsvariablen für die Ausführung des Build-Scripts unter Windows konfiguriert. Die Einstellungen sind direkt in der Datei beschrieben. Das RMC Repository enthält eine Beispielkonfiguration unter `/src/setenv.cmd.sample`

setenv.sh: Hier werden verschiedene Verzeichnisse und Umgebungsvariablen für die Ausführung des Build-Scripts unter Linux konfiguriert. Die Einstellungen sind direkt in der Datei beschrieben. Das RMC Repository enthält eine Beispielkonfiguration unter `/src/setenv.sh`

7.2 Komponenten

Die in diesem Kapitel vorgestellten Komponenten wurden ensprechend Architektur siehe Kapitel 6 übernommen. Teilweise wurde bei der Implementierung auf schon vorhandene Projekte zurückgegriffen. Um dem Entwickler einen schnellen Einstieg zu ermöglichen, folgt nun eine Auflistung, in der die Architekturkomponenten den Implementationen zugeordnet sind.

Tabelle 7.1 listet alle Schnittstellen von RMC auf, die die einzelnen Komponenten nutzen. Diese Komponenten sind in Tabelle 7.2 dargestellt.

Tabelle 7.1: Entwicklung: Übersicht der Schnittstellen

Architektur-schnittstellen	Projektname	Paket
IModel	`de.upb.pg.rmc.umlz`	`de.upb.pg.rmc.umlz`
IRefactoringEditor	`de.upb.pg.rmc.refactoring .editorinterface`	`de.upb.pg.rmc.refactoring .editorinterface`
IRefactoringDiagnostics	`de.upb.pg.rmc.refactoring .diagnosticsinterface`	`de.upb.pg.rmc.refactoring .diagnosticsinterface`
IOCL	`de.upb.pg.rmc.umlz .ocl.oclinterface`	`de.upb.pg.rmc.umlz .ocl.oclinterface`
IMQP	`de.upb.pg.rmc.mqp .diagnosticsinterface`	`de.upb.pg.rmc.mqp .diagnosticsinterface`
IZParser	`de.upb.pg.rmc.umlz`	`de.upb.pg.rmc.umlz.izparser`
IMQPInternalModel	`de.upb.pg.rmc.mqp`	`de.upb.pg.rmc.mqp`

Tabelle 7.2: Entwicklung: Übersicht der Komponenten

Architektur-komponenten	Projektname	Paket
UML/Z-Model	`de.upb.pg.rmc.umlz`	alle Pakete bis auf `de.upb.pg.rmc.umlz` und `de.upb.pg.rmc.umlz.izparser`
UML/Z-Editor	Klassendiagramm: `de.upb.pg.rmc.classeditor`	alle Pakete des Projekts
	Zustandsdiagramm: `de.upb.pg.rmc.stateeditor`	alle Pakete des Projekts
ModelQualityPlan	GUI: `de.upb.pg.rmc.mqp.ui`	alle Pakete des Projekts
	Messungen: `de.upb.pg.rmc.mqp.measurement`	alle Pakete des Projekts
Refactoring	`de.upb.pg.rmc.refactoring`	alle Pakete des Projekts
Diagnostics	`de.upb.pg.rmc.diagnostics`	alle Pakete des Projekts
CZTWrapper	`de.upb.pg.rmc.cztwrapper`	alle Pakete des Projekts
OCLWrapper	`de.upb.pg.rmc.umlz.ocl`	alle Pakete des Projekts

7.2.1 Modell

7.2.1.1 Allgemeines

Generierung der Editoren

Grundlage für die Generierung des UML/Z-Modells ist das EMF-Modell, aus dem der Java-Code generiert wird. Das Modell befindet sich im **model**-Ordner des `de.upb.pg.rmc.umlz` Projekts.

Benötigt werden dazu folgende Dateien:

- UML.ecore: UML/Z-Modell
- UML.genmodel: Generierungsmodell für das UML/Z-Modell

Die automatische Code-Generierung erzeugt die folgenden Pakete:

- `de.upb.pg.rmc.umlz`
- `de.upb.pg.rmc.umlz.edit`
- `de.upb.pg.rmc.umlz.editor`
- `de.upb.pg.rmc.umlz.tests`

7.2.1.2 Durchgeführte Erweiterungen

Das UML/Z-Modell basiert, wie in vorhergehenden Kapiteln bereits erwähnt, auf der UML2-Implementierung der Eclipse-Foundation. Aufgrund der Komplexität der UML2 umfasst die UML2-Implementierung bereits einige manuelle Anpassungen gegenüber dem generierten Modell-Code. Auf diese Änderungen soll im Folgenden nicht eingegangen werden, weil sie bereits so aus der Eclipse UML2-Implementierung übernommen worden sind.

> **Hinweis:** Beim Generieren des Modells sollte man sich aber darüber im klaren sein, dass das Generieren auf Basis des bestehenden Codes geschehen muss. Einem Modell, welches komplett neu ausschließlich aus Ecore und Genmodel erstellt wird, fehlen etliche, für die Grundfunktionalität des Modells erforderliche, manuelle Code-Anpassungen.

Anbindung an den Z-Parser

Um Z-Ausdrücke beim Einlesen einer Modelldatei parsen zu können, müssen komplette Strukturen des Modells unter bestimmten Umständen bereits feststehen, also bereits eingelesen worden sein und als Objekte vorliegen. Begründet ist dies durch die Tatsache, dass beim Parsen der Z-Ausdrücke in der Komponente **CZTWrapper** Modellelemente einzelnen Variablen im Z-Ausdruck hinterlegt werden (siehe dazu [MU05] und die Beschreibung der Architektur in Kapitel 6). Um alle Z-Ausdrücke beim Einlesen des Modells erst zu sammeln und anschließend nach erfolgreichem Einlesen des ganzen Modells zu parsen, wurden das Paket `de.upb.pg.rmc.umlz.editor.z` und die enthaltenen Klassen `ZPredicateSpecificationContainer` und `ZPredicateSpecificationContainerProvider` erzeugt.

Die Klasse `ZPredicateSpecificationContainerProvider` implementiert dabei das Interface `IZPredicateSpecificationContainer` im Paket `de.upb.pg.rmc.umlz.util` der Komponente `de.upb.pg.rmc.umlz`, welches als Erweiterungspunkt[8] `de.upb.pg.rmc.umlz.ZPredicateSpecificationContainer` der Komponente `de.upb.pg.rmc.umlz` angeboten wird. In der Klasse `ZPredicateSpecificationContainer` werden alle Z-Ausdrücke gesammelt und nach dem Einlesen des kompletten Modells mit dem Aufruf der Funktion `parseAll` geparst.

Eingabe von Z-Ausdrücken

Der Eingabedialog für die Z-Ausdrücke ist im Projekt `de.upb.pg.rmc.umlz.editor` realisiert. Im Unterpaket `presentation` befindet sich dazu zum Einen die Klasse `ZEditorComposite`, welche die grafische Benutzeroberfläche für den Eingabedialog enthält. Zum Anderen behinhaltet die Klasse `UMLZEditor` die statische Klasse `UMLPropertyDescriptor`. Von dort aus wird der entsprechende Button zum Anzeigen des Eingabe-Dialogs für **ZExpression**

[8] Siehe dazu die Hilfe zu Extension Points von eclipse.org unter http://help.eclipse.org/help32/index.jsp.

und **ZPredicate** erzeugt. Das Vorgehen entspricht dabei dem in den EMF Recipes[9] beschriebenen.

7.2.1.3 Implementierung neuer Funktionen

Anlegen von Operationen im UML/Z Modell für die Verwendung mit OCL

Die Sprache OCL ist nur für Anfragen an Modelle ausgelegt; im Rahmen dieser Anfragen können aber auch Methoden im Modell aufgerufen werden (z.B. im Modell definierte Methoden, die alle Attribute der aktuellen Klasse sowie aller Oberklassen holen). Um Änderungen im Modell mit OCL durchführen zu können, kann man dementsprechend Methoden im Modell anlegen, die dann Änderungen durchführen.

Zunächst sollte man mit Hilfe des Quellcodes und der UML2-Spezifikation feststellen, wo sich die Modellelemente, die verändert werden sollen, befinden. Dann kann im Ecore eine neue Operation angelegt werden.

Die Ecore Datei mit dem Namen UML.ecore befindet sich im Projekt `de.upb.pg.rmc.umlz` im Ordner **model**.

Anlegen von Operationen

Operationen können über das Kontextmenü des entsprechenden Elements hinzugefügt werden (Abbildung 7.2). Entsprechend können auch Parameter zur so erstellten Operation hinzugefügt werden. Über die Properties-Ansicht können der Name der Operation und Parameter, sowie die Typen der Parameter und ggf. auch ein Rückgabetyp für die Operation eingestellt werden.

Dokumentieren der neu angelegten Operationen und Parameter

Um ein Element zu dokumentieren, muss für dieses Zunächst eine **EAnnotation** angelegt werden (Abbildung 7.3). Diese erhält dann als Source-Attribut gemäß Abbildung 7.4 folgenden Wert:

http://www.eclipse.org/emf/2002/GenModel

Für die so erstellte **EAnnotation** muss nun ein **Details Entry** angelegt werden (Abbildung 7.5). Dieser **Details Entry** erhält als **Key** den Wert **documentation** und als **Value** den Dokumentationstext (Abbildung 7.6).

[9] http://wiki.eclipse.org//EMF/Recipes#Recipe:_Create_your_own_property_editor_in_a_generated_application

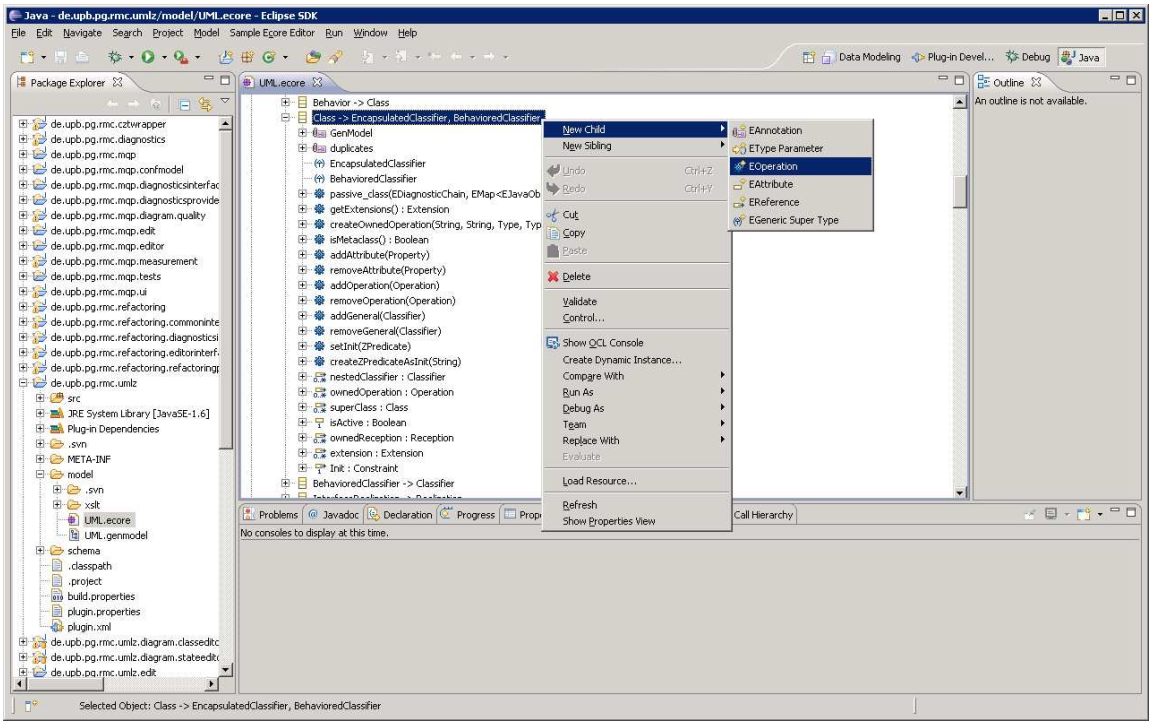


Abbildung 7.2: Modell: Hinzufügen einer Operation

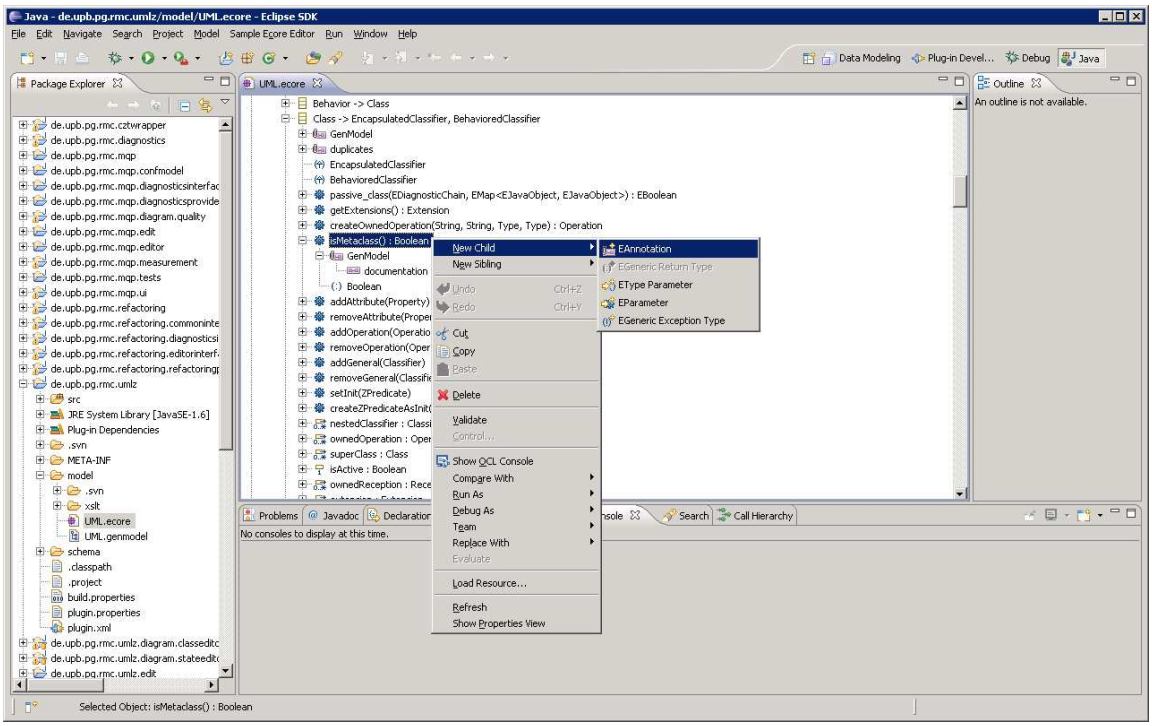


Abbildung 7.3: Modell: Hinzufügen einer EAnnotation

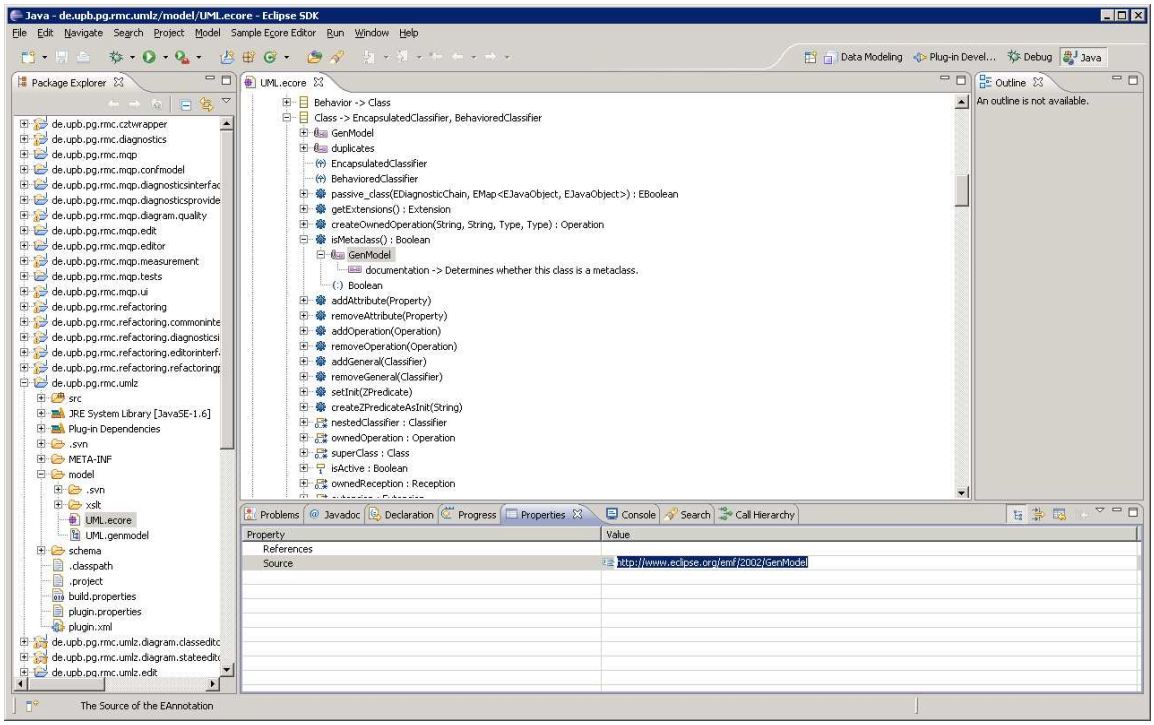


Abbildung 7.4: Modell: Source-Attribut der EAnnotation

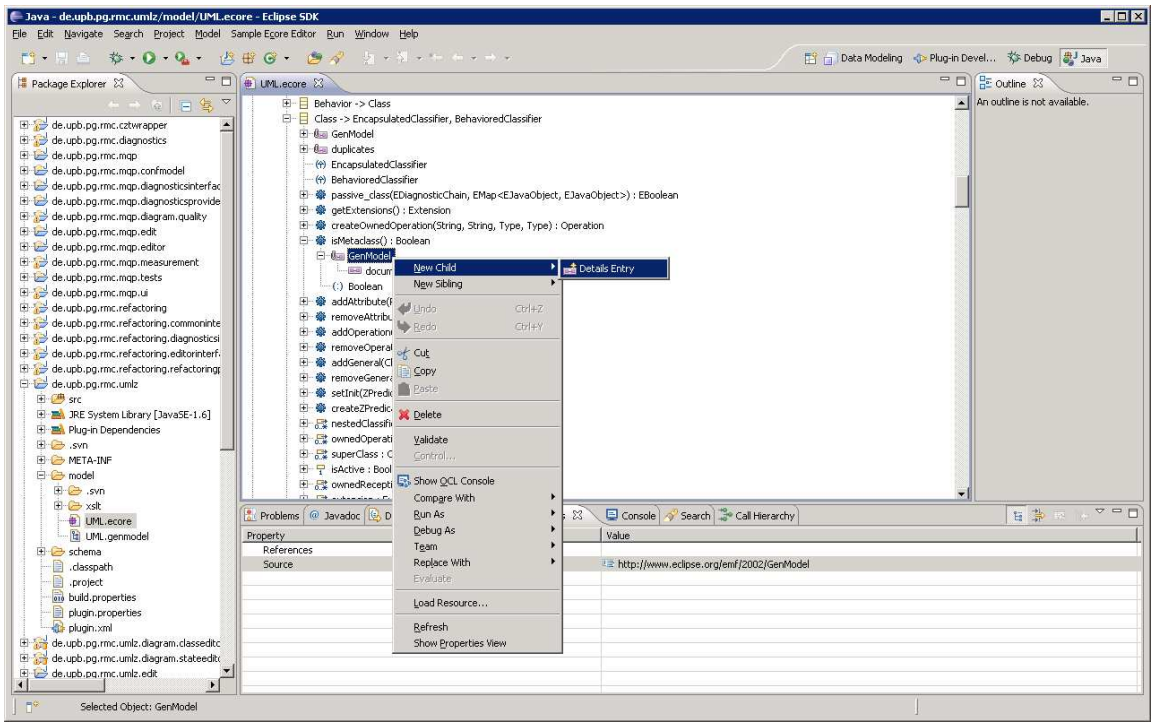


Abbildung 7.5: Modell: Hinzufügen eines Details Entry

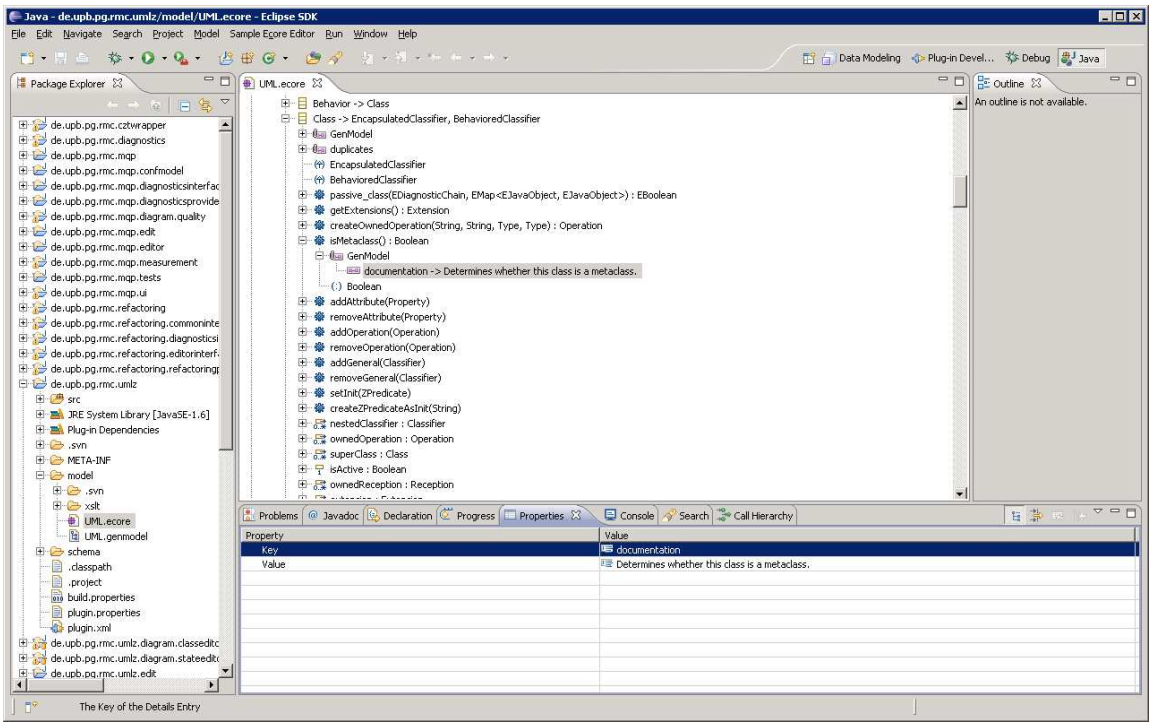


Abbildung 7.6: Modell: Attribute des Details Entry

Anpassen des Genmodels

Das UML/Z-Modell basiert auf dem Eclipse UML2-Modell. Da EMF im Funktionsumfang sehr eingeschränkt ist, wurden von den Entwicklern diverse Modifikationen und Erweiterungen angewandt, um das UML2-Modell mit EMF erstellen zu können. Da diese größtenteils nicht dokumentiert sind, führt dies dazu, dass aus dem UML/Z-Modell kein neues Genmodel erzeugt werden kann. Stattdessen muss das alte Genmodel angepasst werden.

Erstellen eines neuen Genmodels

Hierzu sollte man den modifizierten Ecore zunächst in ein separates Projekt kopieren und dort das Genmodel erstellen.

Dazu wird der Ecore zunächst in ein neues, leeres Projekt kopiert und dort ein neues EMF-Modell erstellt. Dazu wird im ⟨`New Dialog`⟩ in der Kategorie **Eclipse Modeling Framework** der Punkt **EMF Model** gewählt.

Auf der folgenden Seite des Assistenten wird als Dateiname **UML.genmodel** angegeben.

Anschließend wird auf der nächsten Seite **Ecore Model** als Typ des zu importierenden Modells gewählt.

Auf der folgenden Seite wird das Ecore Modell ausgewählt, welches zu diesem Zweck in das neue Projekt kopiert wurde.

Nach einem Klick auf [Next] wird nun zunächst im unteren Bereich des Dialogs **Ecore** als referenziertes Generator-Modell ausgewählt und anschließend oben **umlz** als Root Package.

Nach Abschluss des Assistenten erhält man ein neues Genmodel.

Anpassen des Original-Genmodels

Die Zeilen im so generierten Ecore, die sich auf die neu angelegten Operations beziehen, müssen nun an die entsprechenden Stellen des Original-Genmodels kopiert werden. Hierzu sollte man beide Genmodel-Dateien in einem XML- bzw. Text-Editor öffnen.

Anschließend sollten noch **type**-Attribute aus dem XSI-Namespace gesetzt werden, die im generierten Ecore fehlen. Also z.B. **xsi:type="genmodel:GenOperation"** für Operationen oder **xsi:type="genmodel:GenParameter"** für die Parameter.

Generieren des Modells

Nachdem das Original-Genmodel modifiziert wurde, kann wie üblich das UML/Z-Modell über das Kontextmenü des UML/Z-Elementes in der zweiten Baumhierarchieebene im Genmodel mit **Generate All** neu generiert werden (Abbildung 7.7).

Korrigieren von Fehlern im Modell

Durch das Hinzufügen von zusätzlichen Methoden im Modell kann es zu doppelten Methoden kommen. Das Modell enthält z.B. bereits teilweise die üblichen Java-Setter-Methoden. Diese können aber mit OCL nicht aufgerufen werden, weil sie nicht im Ecore definiert sind. Wenn man nun im Ecore eine solche Methode anlegt und neu generiert, erhält man doppelte Methoden im Java-Code. Die selbst angelegte, leere Methode kann dann einfach gelöscht werden. Die Aufrufe über OCL verwenden dann die schon vorhandene.

Implementieren der neu angelegten Methoden

Wird nicht nur ein Ecore-Eintrag für eine bereits schon bestehende Methode erstellt, muss die neu angelegte Methode nach dem Generieren des Modells noch angepasst werden. Die entsprechende Platzhaltermethode dafür befindet sich im Package `de.upb.pg.rmc.umlz.internal.operations` in der entsprechenden Klasse, zu der die Methode hinzugefügt wurde, also z.B. für die Klasse **NamedElement** in **NamedElementOperations.java**.

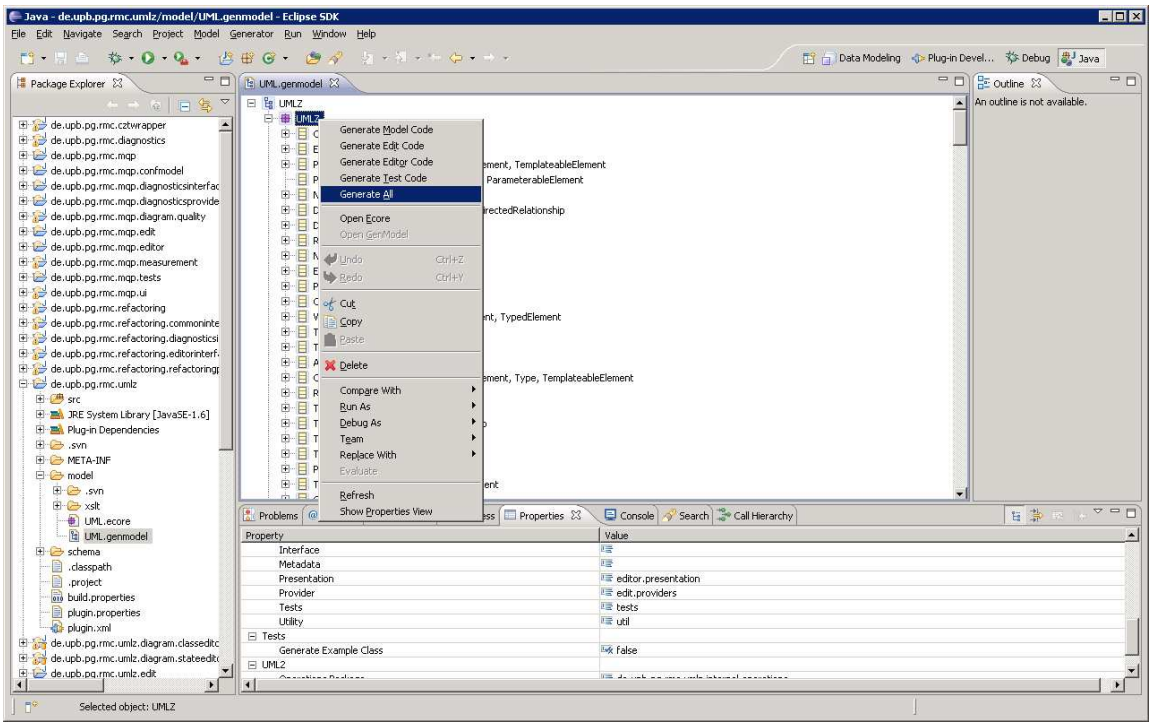


Abbildung 7.7: Modell: Generieren des UML/Z-Modell-Java-Codes

7.2.2 Editor

7.2.2.1 Allgemeines

Generierung der Editoren

Grundlage für den Klassendiagrammeditor und den Zustandsdiagrammeditor sind die Modelle, mit denen GMF den Quelltext generiert . Die Modelle befinden sich im **model** Verzeichnis in den beiden Projekten `de.upb.pg.rmc.umlz.diagram.classeditor` und `de.upb.pg.rmc.umlz.diagram.stateeditor`. Basis für diese Modelle stellen die UML2 Tools dar. Da diese jedoch auf einem UML-Metamodell arbeiten, waren Anpassungen an das UML/Z-Modell notwendig. Jedoch sind die verwendeten Grafiken und große Teile des zusätzlich notwendigen Quelltextes übernommen.

Für grundlegende Änderungen ist oft ein Editieren der Modelle nötig; ebenso bei größeren Änderungen am UML/Z-Modell[10].

Benötigt werden dazu folgende Modelle und Template-Dateien:

- Klassendiagrammeditor

  UML.ecore: UML/Z-Metamodell

[10] Wenn im UML/Z-Modell neue Operationen für den Zugriff mit OCL erstellt wurden, ist ein Generieren nicht notwendig.

UML.genmodel: Generierungsmodell für das UML/Z-Modell
umlz2.gmfgraph: die grafischen Elemente des Editors
umlz.gmftool: die Palette des Editors
umlz2.gmfmap: das Mapping der vorherigen Modelle
umlz2.gmfgen: resultierendes Modell für die Generierung
templates Verzeichnis: Templates für die Generierung

- Zustandsdiagrammeditor

UML.ecore: UML/Z-Metamodell
UML.genmodel: Generierungsmodell für das UML/Z-Modell
propsheet.ecore: Modell für die Erweiterung der **Eclipse Tabbed Properties View**
propsheet.genmodel: Modell für die Erweiterung der **Eclipse Tabbed Properties View**
stateMachineDiagram.gmfgraph: die grafischen Elemente des Editors
activityDiagram_controlNodes.gmfgraph: die grafischen Elemente des Editors
stateMachineDiagram.gmftool: die Palette des Editors
stateMachineDiagram.gmfmap: das Mapping der vorherigen Modelle
stateMachineDiagram.gmfgen: resultierendes Modell für die Generierung
templates Verzeichnis: Templates für die Generierung
gmf-graph Verzeichnis: Templates für die Generierung
codegen Verzeichnis: Templates für die Generierung

Abbildung 7.8 zeigt den Ablauf der Erzeugung des Quelltextes. Die oben erwähnten Modelle dienen als Eingabe für das GMF-Framework.

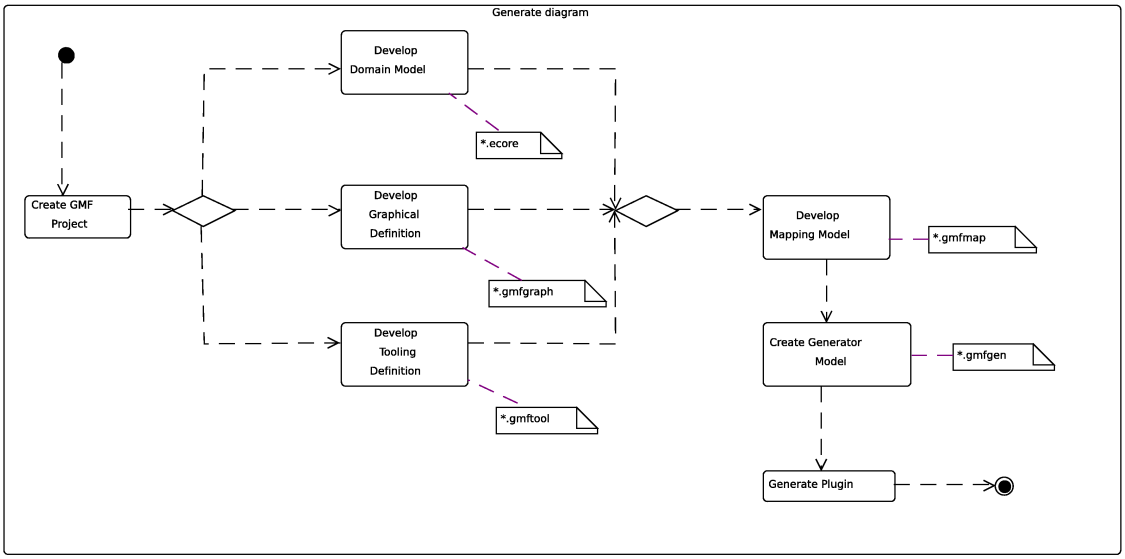


Abbildung 7.8: Schematischer Ablauf der Generierung des Editors

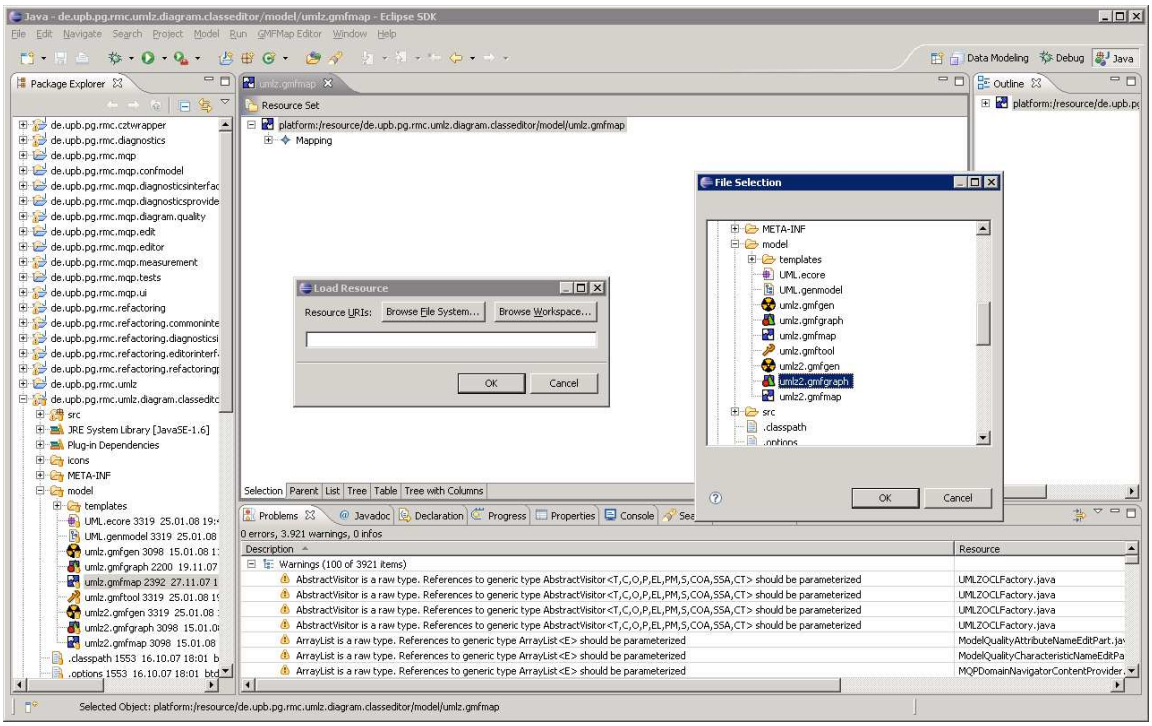


Abbildung 7.9: Editor: GMF-Mappingmodel um Ressourcen erweitern

Aus den Modellen für die grafischen Elemente, die Toolbar und dem Metamodell wird zunächst ein Modell für das Mapping dieser Modelle erstellt. Abbildung 7.9 zeigt den Dialog, mit dem Ressourcen zu einem Modell hinzugefügt werden. Bei der Neuerstellung eines Mappings werden diese Informationen direkt durch den **Wizard** abgefragt.

Aus diesem Mapping wird das Modell für die Generierung erstellt. Der Anwender wählt auf der **gmfmap**-Ressource im ⟨`Kontextmenü`⟩→⟨`Create Generator Model`⟩, dargestellt in Abbildung 7.10. Dieses erzeugt das **Genmodel**.

Das **Genmodel** enthält die Informationen, die GMF für die Generierung des Quelltextes der Editoren benötigt. Erstellt werden diese durch Auswahl von ⟨`Generate Diagram Code`⟩ auf dem Wurzelelement des **Genmodels**, dargestellt in Abbildung 7.11.

Weitere Informationen zu diesem Thema finden sich auch der GMF-Webseite. Der erzeugte Quelltext ist nur zusammen mit dem generierten Modell nutzbar. Benötigt werden folgende Pakete:

- `de.upb.pg.rmc.umlz`
- `de.upb.pg.rmc.umlz.edit`
- `de.upb.pg.rmc.umlz.editor`

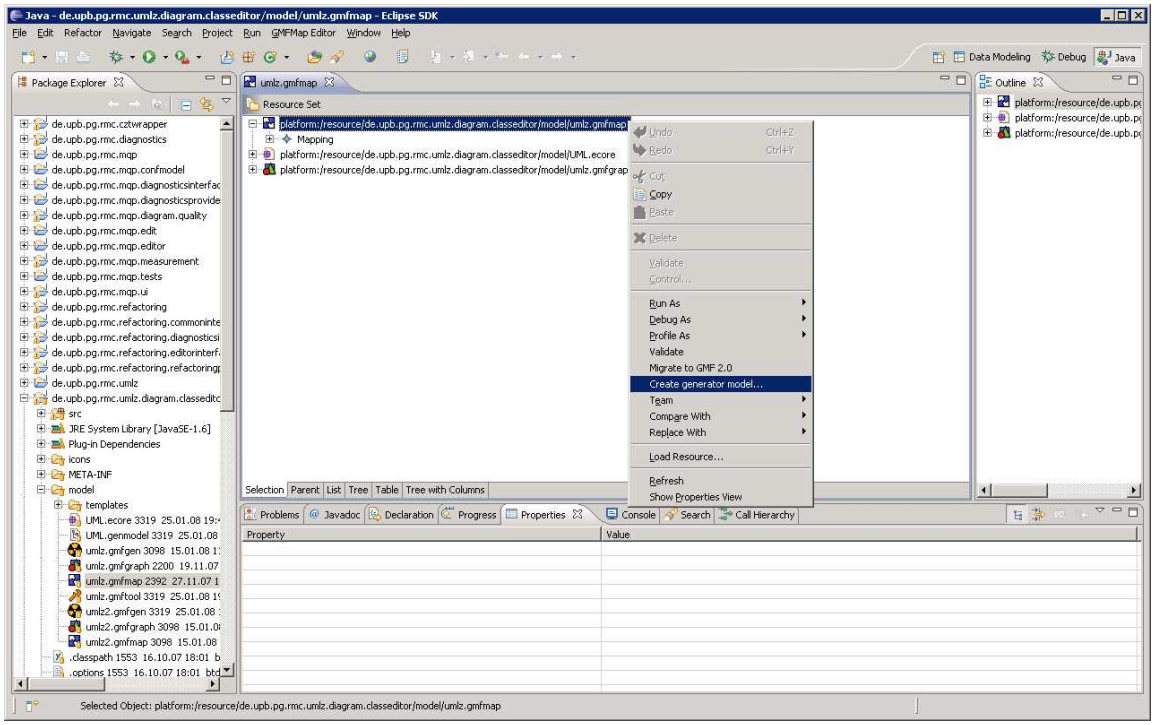


Abbildung 7.10: Editor: GMF Genmodel erstellen

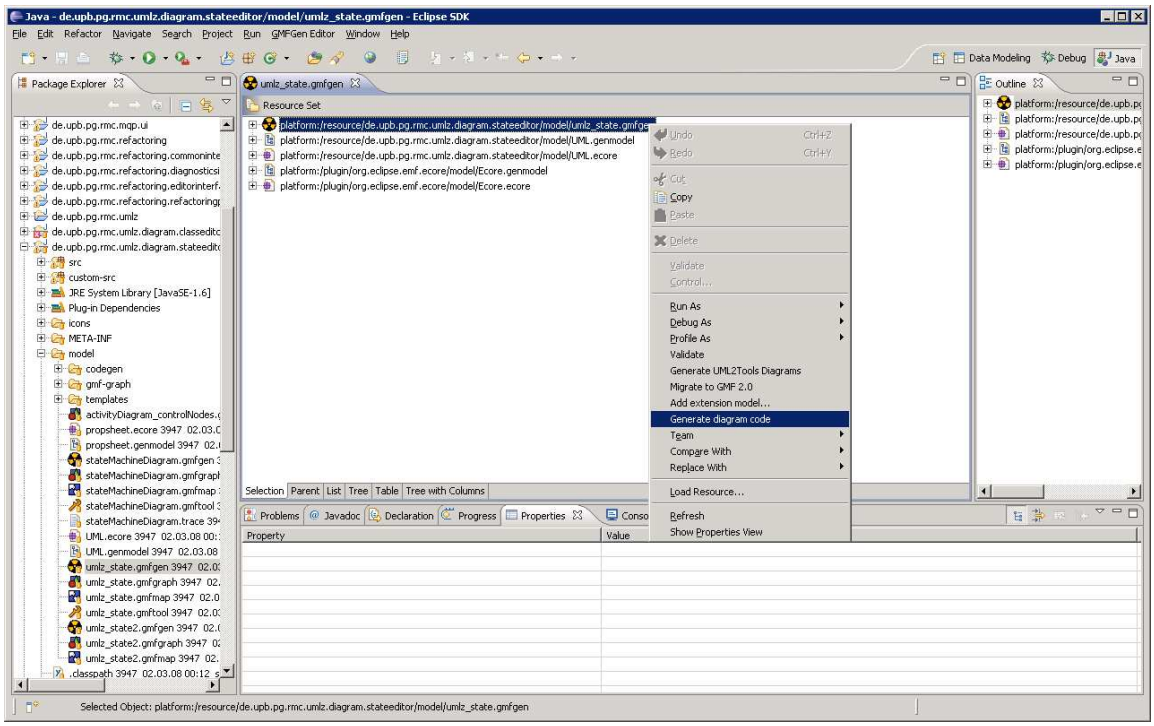


Abbildung 7.11: Editor: GMF-Quelltext für Diagramme generieren

Zusätzliches

Der so entstandene Quelltext ist lauffähig, jedoch sind viele Funktionen nicht vollständig implementiert oder UML/Z-konform. Die UML2 Tools bieten einen umfangreichen Editor für UML2-Modelle. Trotz der Unterschiede von UML2 und UML/Z ließen sich viele Implementierungen übernehmen. So sind die **Parser** vollständig übernommen worden. Die vorgenommenen Änderungen sowohl am generierten Quelltext als auch an den UML2 Tools Quelltext finden sich in Kapitel 7.2.2.2.

> **Hinweis:** Methoden und Klassen mit Änderungen werden vor dem Überschreiben durch Generierung geschützt mit:
>
> - @NOT-GENERATED
> - @GENERATED-NOT
> - @GENERATED NOT
>
> Gleichzeitig ist dies eine Möglichkeit die per Hand implementierten Methoden im umfangreichen Quelltext zu finden.

Die verwendeten Grafiken stammen ebenfalls aus den UML2 Tools. Weiterhin werden Grafiken des Modelles genutzt. In den Projekten liegen sie unter:

- de.upb.pg.rmc.umlz.diagram.classeditor/icons/
- de.upb.pg.rmc.umlz.diagram.stateeditor/icons/
- de.upb.pg.rmc.umlz.edit/icons/

7.2.2.2 Durchgeführte Erweiterungen

Für die speziellen Anforderungen an den UML/Z-Editor waren einige Anpassungen und Implementierungen notwendig. Das folgende Kapitel geht auf diese Erweiterungen ein und erläutert die durchgeführten Änderungen.

Klassendiagrammeditor

Parser

Im Bereich der Parser sind zwei wesentliche Bereiche zu betrachten. Es müssen die per Hand implementierten Parser mit dem generierten Quelltext kombiniert werden. Dazu ist es notwendig, den Editor über die entsprechenden Klassen zu informieren. Dies geschieht über die Klasse `UMLZParserProvider` im Paket `de.upb.pg.rmc.umlz.diagram.classeditor.providers`. Hier finden sich die Elemente des UML/Z-Modells, die über den Editor angelegt werden können. Für Elemente, die spezialisierte Parser nutzen, müssen die Methoden

`createElementParser` und `getElementParser` implementiert werden. Die Zuordnung ist nach Neugenerierung des Quelltextes zu überprüfen und gegebenenfalls zu korrigieren.

Weiterhin erfolgte folgende Änderung an den Parsern:

Im erzeugten Editor erhalten Methoden und Attribute, die mit **public** gekennzeichnet sind, kein + als Symbol. Dies wurde geändert, so dass auch öffentlich sichtbare Elemente explizit gekennzeichnet werden. Diese Implementierungen finden sich in `AbstractToString` im Paket `de.upb.pg.rmc.umlz .diagram.classeditor.parser`. Für die Beschriftung von Attributen an Assoziationen folgt daraus direkt die Notwendigkeit einer Überprüfung auf leere **Labels**, damit Sichtbarkeitsattribute nicht angezeigt werden, falls ein Label leer ist. Zu finden ist dies in der Klasse `AssociationEndToString` im Paket `de.upb.pg.rmc.umlz .diagram.classeditor.parser.association.end`.

Zustandsdiagramme auf Klassen erzeugen

Es soll möglich sein, Zustandsdiagramme über das Kontextmenü von Klassen auf dem Canvas zu erstellen. Dazu wurde die plugin.xml Datei des **de.upb.pg.rmc.umlz.diagram .stateeditor** Projekts um folgenden Eintrag erweitert:

```
<extension point="org.eclipse.ui.popupMenus">
  <?gmfgen generated="false"?>
  <objectContribution
    id="de.upb.pg.rmc.umlz.diagram.classeditor.InitDiagram"
    objectClass="de.upb.pg.rmc.umlz.diagram.classeditor.edit.parts
      .ClassEditPart">
    <action
      label="initDiagramActionLabel"
      class="de.upb.pg.rmc.umlz.diagram.stateeditor.part
        .UMLZInitStateDiagramFileAction"
      menubarPath="additions"
      enablesFor="1"
      id="de.upb.pg.rmc.umlz.diagram.classeditor.InitDiagramAction">
    </action>
  </objectContribution>
</extension>
```

Listing 7.1: Kontextmenü von Klassen auf dem Canvas um Eintrag für Zustandsdiagramme erweitern

Das Anlegen des neuen Diagrammes geschieht dann in der Klasse `UMLZInitStateDiagram-FileAction`. Diese stellt eine Modifikation der Klasse `UMLZInitDiagramFileAction` dar, mit der Diagramme über den **Project Explorer** auf Modelldateien erzeugt werden.

Outline

Folgende Elemente des UML/Z-Modells sind nicht direkt auf dem Canvas sichtbar:

- Type Declaration
- ZExpression Specification
- ZPredicate Specification

Folglich wird eine Möglichkeit benötigt, solche Elemente anlegen und bearbeiten zu können. Dies wurde durch einen zusätzlichen Baumeditor, analog zu dem in EMF enthaltenen, realisiert. Zu finden ist dieser Editor in der Outline des UML/Z-Editors. Die Umsetzung des Baumes und die darauf realisierten Funktionen erläutert dieser Abschnitt. Informationen über die daraus resultierenden notwendigen weiteren Änderungen liefern die beiden folgenden Abschnitte.

Die Umsetzung für den Baumeditor findet sich in der Klasse `UMLZDiagramEditor` im Paket `de.upb.pg.rmc.umlz.diagram.classeditor.part`, siehe Diagramm 6.16 der Architekturdokumentation. Zu erwähnen sind folgende Bereiche:

EMF Tree: Der dargestellte Baum des Modelles ist eine Implementierung analog zu dem Treeeditors des UML/Z-Modells. Die Elemente können mit der Maus verschoben werden. Besonderheit dieses Baumes ist das selektive Wurzelelement. Bei der Initialisierung der Outline wird ein **Listener** auf das grafische Modell gesetzt und Selektionen von Elementen auf den Canvas werden direkt als Wurzel des Baumes gewählt. Dies wird durch den Aufruf von `contentOutlineViewer.setSelection(new StructuredSelection(selectedDomainElement))` in der Methode `getContentOutlinePage` umgesetzt. Die **Provider** für den Tree finden sich in den Klassen: `UMLPropertySource`, `UMLPropertyDescriptor` und `UMLAdapterFactoryContentProvider`. Für dessen genauere Funktionsweise und Implementierung sei auf das UML/Z-Modell und die UML2 Dokumentation verwiesen.

RMC Menüpunkt: In einem normalen EMF-Baumeditor sind die Kontexmenüeinträge ⟨`Create child`⟩ und ⟨`Create sibling`⟩ vorhanden. Diese sind jedoch sehr unübersichtlich und unkomfortabel. Aus diesem Grund wurde das Kontextmenü um das ⟨`RMC`⟩-Menü ergänzt. In diesem finden sich nur die wichtigsten Elemente des Modelles. Dieses Menü stellt eine gefilterte Sicht auf die beiden Menüs **Create child** und **Create sibling** dar. Die angezeigten Elemente finden sich in der Collection **myActionList**.

delete Menüpunkt: Das Menü zum Löschen von Elementen des Baumes wurde auch neu implementiert. Das Entfernen von Elementen wird durch den Aufruf eines **DeleteCommand** über eine **Action** durchgeführt. Details zu diesen Techniken finden sich in den Dokumentationen über Eclipse und EMF.

Initialize Statemachine Menüpunkt: Auf Klassen soll es möglich sein, Zustandsdiagramme zu erzeugen. Der Kontextmenü wurde um einen solchen Eintrag erweitert. Dazu wurde die plugin.xml Datei des **de.upb.pg.rmc.umlz.diagram.stateeditor** Projekts um folgenden Eintrag erweitert:

```
<extension point="org.eclipse.ui.popupMenus">
  <?gmfgen generated="false"?>
  <objectContribution
    id="de.upb.pg.rmc.umlz.diagram.classeditor.InitDiagram"
```

```
        objectClass="de.upb.pg.rmc.umlz.internal.impl.ClassImpl">
        <action
          label="initDiagramActionLabel"
          class="de.upb.pg.rmc.umlz.diagram.stateeditor.part
            .UMLZInitStateDiagramFileAction"
          menubarPath="additions"
          enablesFor="1"
          id="de.upb.pg.rmc.umlz.diagram.classeditor
            .InitDiagramAction">
        </action>
      </objectContribution>
    </extension>
```

Listing 7.2: Kontextmenü von Klassen in der Outline um Eintrag für Zustandsdiagramme erweitern

Das Anlegen des neuen Diagrammes geschieht dann in der Klasse `UMLZInitStateDiagramFileAction`. Diese stellt eine Modifikation der Klasse `UMLZInitDiagramFileAction` dar, mit der Diagramme über den **Project Explorer** auf Modelldateien erzeugt werden.

Properties

Durch die Nutzung des EMF-basierten Treeeditors im GMF-Editor existieren zwei Typen von Elementen: EMF-Elemente und GMF-Elemente, wobei die Domänenelemente des GMF-Teiles eine Untermenge der EMF-Domänenelemente darstellen. Aus diesem Grund war es notwendig die **PropertyView** zu modifizieren. Genutzt wird jetzt die Implementierung der EMF Properties. Weiterhin wurde diese, um die Möglichkeit den **Z-Editor** zu starten, erweitert.

Zustandsdiagrammeditor

Parser

Ein großer Teil der Parser konnte von den UML2 Tools übernommen werden. Da jedoch in UML/Z keine **Trigger** und **Guard** Attribte an einer Transition vorhanden sind, waren hier Anpassungen notwendig.

In der Klasse `TransitionParser` im Paket `de.upb.pg.rmc.umlz.diagram.stateeditor.parser` müssen die Methoden `getPrintString` und `isAffectingEvent` neu implementiert werden:

```
public String getPrintString(IAdaptable element, int flags) {
  EObject eObject = (EObject)element.getAdapter(EObject.class);
  if (eObject instanceof Transition) {
    Transition transition = (Transition) eObject;
    StringBuffer printStringBuffer = new StringBuffer(20);
```

```
    Behavior effect = transition.getEffect();
    if (effect != null) {
      if (transition.getName() != null) {
        printStringBuffer.append(transition.getName());
      }
    }

    return printStringBuffer.toString();
  }
  return "";
}
```

Listing 7.3: Implementierung der getString-Methode im Parser für Transitionen

```
public boolean isAffectingEvent(Object event, int flags) {
  if (event instanceof Notification) {
    Object feature = ((Notification) event).getFeature();
    UMLZPackage.eINSTANCE.getTransition_Effect().equals(feature);
  }
  return false;
}
```

Listing 7.4: Implementierung der isAffectingEvent-Methode im Parser für Transitionen

Transitionen

Um Transitionen Operationen zuzuordnen, wurde die **stateMachineDiagram.gmfgen** um die Möglichkeit erweitert, das **PropertySheet** von GMF anzupassen. Dazu ist es notwendig, die Modelle **propsheet.genmodel** und **propsheet.ecore** in das Genmodell zu laden. Dies geschieht wie in Kapitel 7.2.2.1 beschrieben. Weitere Informationen zu diesem Thema finden sich in diesem Artikel [PRO].

Mit dieser Vorbereitung wird zunächst im Genmodell unter **de.upb.pg.rmc.umlz.diagram.stateeditor.sheet** ein neuer Eintrag für einen **Custom Property Tab** angelegt. Diesem werden über einen **Typed selection filter** die Transitionen zugeordnet. Abbildung 7.12 zeigt die Zuordnung im Modell.

Dies hat den Effekt, dass bei der Selektion einer Transition im Canvas der zusätzliche **Tab** in den Properties angezeigt wird. Nun muss diese Seite noch mit Inhalt gefüllt werden. Dazu wird ein neues Wurzelelement vom Typ **Property Section Container** angelegt. Dieses erhält die notwendigen Kinder, um die Seite für die Properties zu generieren. Diese enthält dann schon alle notwendigen Texte und die **Combobox** für die Auswahl der Operationen. Abbildung 7.13 zeigt die entsprechenden Einträge im Modell.

Der generierte Quellcode muss noch für den Zugriff auf das UML/Z-Modell erweitert werden. Dies geschieht in der Klasse `UMLZTransitionPropertySection` im Paket `UMLZTransitionPropertySection`. Der generierte Quelltext enthält bereits Methoden für den korrekten Zugriff auf das Modell mittels **Commands**. Zu implementieren sind die `getter`- und

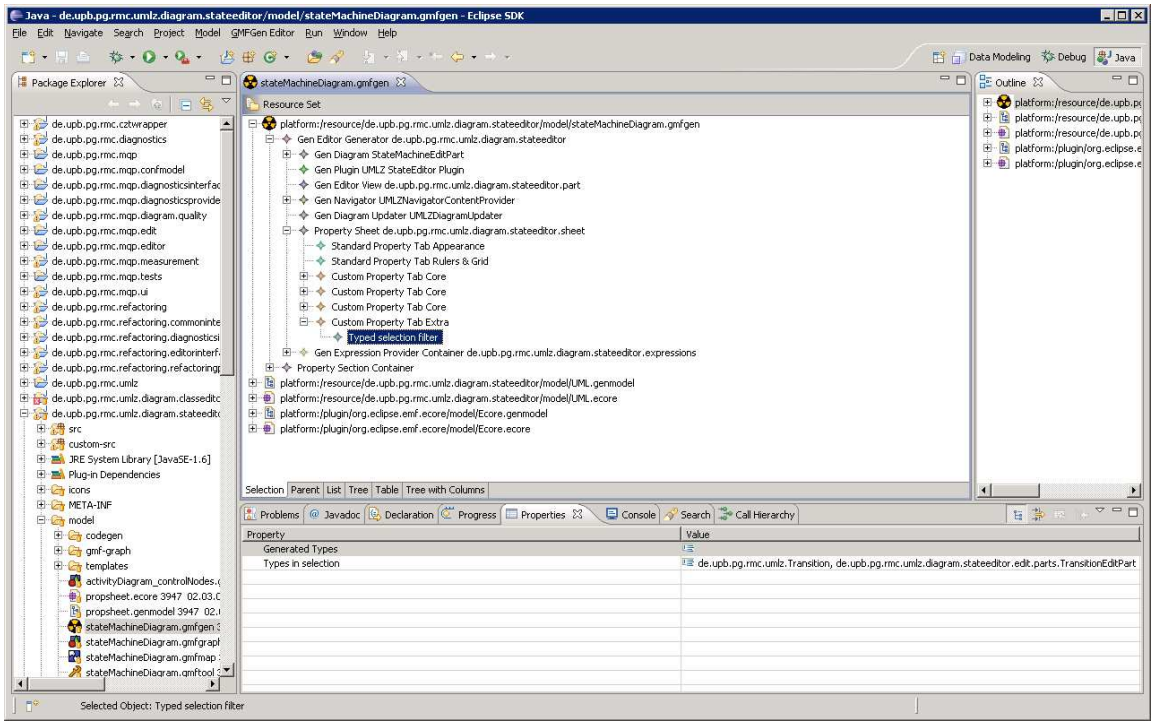


Abbildung 7.12: GMF Genmodel Tab erweitern

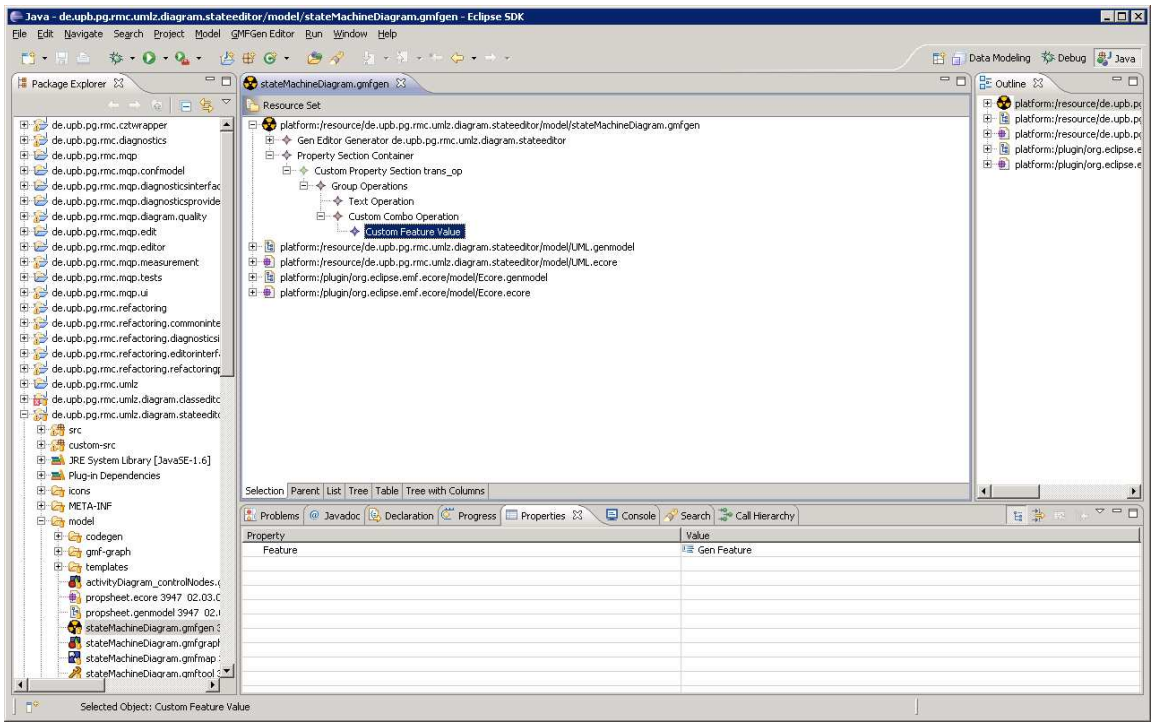


Abbildung 7.13: GMF Genmodel Tab erweitern

`setter`-Methoden für die Transitionen und das Setzen des korrekten Labels der Transition, damit die gewählte Methode auf der Zeichenfläche als Kantenbeschriftung erscheint.

7.2.2.3 Implementierung neuer Funktionen

Der UML/Z-Editor unterstützt schon eine große Anzahl an Funktionen. Denoch bestehen noch Kapazitäten für die Erweiterung. Dieses Kapitel soll einige Anregungen für mögliche Ergänzungen liefern.

Elemente im RMC Menü

Das ⟨`RMC`⟩-Menü im Klasseneditor bietet eine gefilterte Sicht auf die Menüs ⟨`Create child`⟩ und ⟨`Create sibling`⟩. Um weitere Elemente in dieses Menü aufzunehmen, müssen diese im Quelltext ergänzt werden[11]. In der Klasse `UMLZDiagramEditor` im Paket `de.upb.pg.rmc.umlz.diagram.classeditor.part` findet sich die Collection **fillMyActionList**. In dieser werden die Namen der Elemente des RMC-Menüs verwaltet. Für weitere Einträge muss die Collection einfach ergänzt werden. Die entsprechenden Namen finden sich in der **UML.ecore** des UML/Z-Modells.

Z-Editor in RMC Menü

Eine weitere mögliche Erweiterung stellt die Ergänzung des ⟨`RMC`⟩-Menüs um einen Eintrag für den Z-Editor dar. Der Anwender müsste dann nicht aus dem Menü in die Properties wechseln, um den Z-Editor zu starten. Um dies zu implementieren, muss die Operation `fillContextMenu` in der Klasse `UMLZDiagramEditor` im Paket `de.upb.pg.rmc.umlz.diagram.classeditor.part` erweitert werden. Ein erster Ansatz ist die Ergänzung um folgenden Quelltext:

```
//add z-Editor to contextmenu if Z-Element is selected in outline
Object treeSelection =
  ((StructuredSelection)this.getContentOutlinePage().getSelection())
  .getFirstElement();
  if (treeSelection instanceof ZExpressionSpecificationImpl ||
    treeSelection instanceof ZPredicateSpecificationImpl) {
    MenuManager zEditorMenu = new
      MenuManager(UMLZDiagramEditorPlugin.getInstance()
      .getString("customFilteredContextMenuLabel"));
    zEditorMenu.add(new Action("Z-Editor") {
      @Override
      public void run() {

        final Object object =
          ((StructuredSelection)UMLZDiagramEditor.this
```

[11] Nur Elemente die sich bereits in den beiden erwähnten Menüs befinden lassen sich auf diese Art ergänzen. Für weitere Elemente ist eine Anpassung des UML/Z-Modells notwendig.

```
    .getContentOutlinePage().getSelection()).getFirstElement();
  if (object instanceof
    de.upb.pg.rmc.umlz.internal.impl.ZPredicateSpecificationImpl)
    {
    final ZEditorComposite zec[] = new ZEditorComposite[1];
    Dialog d = new
      Dialog(UMLZDiagramEditor.this.getSite().getShell()) {
      //TODO
      @Override
      protected Control createDialogArea(Composite parent) {

        zec[0] = new ZEditorComposite(parent, 0);
        zec[0].setZPredicateSpecification(
          (ZPredicateSpecification) object);
        return zec[0];
      }

      protected void okPressed() {
        ZPredicateSpecification ze = (ZPredicateSpecification)
          object;
        IPredWrapper result = ze.tryParse(zec[0].getZ());
        if (result != null) {
          zec[0].setPredWrapper(result);
          super.okPressed();
        }
      }
    };
    d.create();
    d.getShell().setText("ZPredicate Editor");
    d.setBlockOnOpen(true);
    d.open();
    if (d.getReturnCode() == Dialog.OK) {
      // TODO use command for change!
      ((ZPredicateSpecification) object).setZPredicate(new
        ZPredicate(zec[0].getPredWrapper()));

    }
  }
  if (object instanceof de.upb.pg.rmc.umlz.internal.impl
    .ZExpressionSpecificationImpl) {
    final ZEditorComposite zec[] = new ZEditorComposite[1];
    Dialog d = new
      Dialog(UMLZDiagramEditor.this.getSite().getShell()) {

      @Override
      protected Control createDialogArea(Composite parent) {

        zec[0] = new ZEditorComposite(parent, 0);
        zec[0].setZExpressionSpecification(
          (ZExpressionSpecification) object);
        return zec[0];
      }
```

```
            protected void okPressed() {
              ZExpressionSpecification ze = (ZExpressionSpecification)
                object;
              IExprWrapper result = ze.tryParse(zec[0].getZ());
              if (result != null) {
                zec[0].setExprWrapper(result);
                super.okPressed();
              }
            }
          };
          d.create();
          d.getShell().setText("ZExpression Editor");
          d.setBlockOnOpen(true);
          d.open();
          if (d.getReturnCode() == Dialog.OK) {
            // TODO use command for change!
            ((ZExpressionSpecification) object).setZExpression(new
              ZExpression(zec[0].getExprWrapper()));
          }
        }
      }
    });
  mgr.add(zEditorMenu);
}
```

Listing 7.5: Mögliche Implementierung für den Aufruf des Z-Editors aus dem RMC-Menü

Diese Implementierung in Listing 7.5 stellt nur eine rudimentäre Lösung dar. Die Z-Ausdrücke werden hierbei nicht über **Commands** im Modell gespeichert, sondern direkt gesetzt. Dieses Vorgehen ermöglicht jedoch keine Undo/Redo-Aktionen und kann zu Komplikationen führen. Der Quelltext sollte an den mit **TODO** gekennzeichneten Stellen entsprechend erweitert werden.

Neue Diagrammtypen

UML/Z besteht aus Diagrammen vom Typ Klassendiagramm und Zustandsdiagramm. Falls hier eine Erweiterung des Modells um einen neuen Typ stattfinden sollte, muss der UML/Z-Editor um die entsprechenden Funktionen ergänzt werden. Dazu sind folgende Schritte notwendig:

- ein Modell für die Notation des neuen Diagrammes erstellen (`*.gmfgraph`)
- die Toolbar für dieses Diagramm modellieren (`*.gmftool`)
- aus UML/Z-Modell, Graph-Modell und Tool-Modell ein Mapping erstellen (`*.gmfmap`)
- aus diesem Mapping das Modell für die Generierung erstellen (`*.gmfgen`)
- den Quelltext generieren und notwendige Anpassungen vornehmen

Diese Schritte verlaufen analog zu der Beschreibung der Erstellung des aktuellen UML/Z-Editors in Kapitel 7.2.2.1. Um den neu erstellten Editor in der passenden **RMC**-Gruppe im

Eclipse-Wizard erscheinen zu lassen[12], ist es notwendig, die **plugin.xml** anzupassen. Im **ExtensionPoint** `<extension point="org.eclipse.ui.newWizards">` wird der Eintrag für **category** auf `"rmc"` gesetzt. Der neue Diagrammtyp kann nun in der Anwendung ausgewählt und genutzt werden.

7.2.3 Modellqualitätsplan

7.2.3.1 Änderungen am Modell

Generierung des Modell-Quelltexts

Grundlage für die Generierung des MQP-Modells ist das EMF-Modell, aus dem der Java-Code generiert wird. Das Eclipse Modeling Framework (EMF) [EMF] ist ein Open-Source Java-Framework zur automatisierten Erzeugung von Quellcode anhand von strukturierten Modellen. Das Modell befindet sich im **model**-Ordner des `de.upb.pg.rmc.mqp` Projekts.

Benötigt werden dazu folgende Dateien:

- `MQP.ecore`: MQP-Modell
- `MQP.genmodel`: Generierungsmodell für das MQP-Modell

Die automatische Code-Generierung erzeugt die folgenden Pakete:

- `de.upb.pg.rmc.mqp`
- `de.upb.pg.rmc.mqp.edit`
- `de.upb.pg.rmc.mqp.editor`
- `de.upb.pg.rmc.mqp.tests`

Erweitern des Metamodells

Um Änderungen am MQP-Metamodell realisieren zu können, müssen die entsprechenden Änderungen in der Ecore-Datei des MQP (**MQP.ecore**) vorgenommen werden. Das Vorgehen entspricht dabei dem generellen Vorgehen beim Ändern/Erstellen von EMF-Modellen, zu dem es zahlreiche Tutorials[13] gibt.

Das Eclipse EMF-Framework ist nicht in der Lage die durchgeführten Änderungen automatisch im Genmodel zu übernehmen. Hier gibt es zwei Möglichkeiten:

1. **Das Genmodel neu zu generieren**: Hierbei müssen anschließend die Code-Generierungseigenschaften im Genmodel (Properties der oberen beiden Baumelemente des Genmodels) aus dem ursprünglichen Genmodel wieder übernommen werden.

[12] Neu generierte Editoren erscheinen im Normalfall unter einem **Example**-Eintrag.
[13] http://www.eclipse.org/modeling/emf/docs/

2. **Das bestehende Genmodel ändern**: Dies ist im Allgemeinen einfacher. Zunächst generiert man ebenfalls ein neues Genmodel in ein separates Projekt und fügt dann mit Hilfe eines Diff-Tools die Änderungen in das Original-Genmodel ein, ohne dabei die Code-Generierungseigenschaften im oberen Bereich der Datei zu übernehmen.

Das unter Punkt 2 beschriebene Vorgehen ähnelt dem in Kapitel 7.2.1.3 für UML/Z beschriebenen.

7.2.3.2 GUI des Editors

RMC wurde als Ecplise-Plugin entwickelt. Daher wurde die GUI des MQP-Editor mit Hilfe des SWT-Frameworks entwickelt. Für einige Elemente wurde zusätzlich das Framework **JFace** und **Eclipse Forms** verwendet. In diesem Kapitel wird der genaue Aufbau der grafischen Oberflächen des MQP-Editors erklärt, und wie man die Oberfläche erweitern kann. Der Quellcode dafür befindet sich im Paket `de.upb.pg.rmc.mqp.ui` im Ordner **src/de/upb/pg/rmc/mqp/ui**.

Aufbau der GUI

Ein MQP besteht aus fünf Modellen. Daher wurden im MQP-Editor fünf Reiter eingeführt. Jeder Reiter repräsentiert ein Modell. Für jeden Reiter gibt es eine `ModelEditorPart`-Klasse (Name des Modells + `ModelEditorPart.java`, z. B.: `InformationNeedModelEditorPart.java`). Soll ein neuer Reiter hinzugefügt oder ein bestehender gelöscht werden, geschieht dies in der Klasse `MQPEditor` im Paket `de.upb.pg.rmc.mqp.editor.presentation` in der Methode `createPages`. Hier werden die Reiter über folgenden Beispielcode angelegt:

```
informationNeedModelEditorPart = new
  InformationNeedModelEditorPart(this);
pageIndex = addPage(informationNeedModelEditorPart, getEditorInput());
setPageText(pageIndex, getString("_UI_2InformationNeedModel"));
```

Listing 7.6: MQP: EditorPart

Dieser Beispielcode legt den Reiter für das Informationsbedürfnismodell an. Die Klasse `informationNeedModelEditorPart` ist die Grundlage für den Reiter **Information Need Model**. In ihr wird die `Form`-Klasse für das Modell mit den grafischen Elementen eingebunden. Über **addpage** werden die Reiter hinzugefügt. Die Reihenfolge, in der die Reiter hinzugefügt werden, entspricht dabei auch der Reihenfolge, in der die Reiter später in Eclipse erscheinen. Soll für ein neues Modell ein neuer Reiter erstellt werden, kann man die Datei **informationNeedModelEditorPart.java** kopieren und muss sie an vier Stellen anpassen:

1. Klassenname
2. Konstruktorname

3. In der Methode `setInput` muss der Befehl viewer.setInput(((ModelQualityPlan)((ResourceSet)input).getResources().get(0).getContents().get(0)).getInformationNeedModel()) angepasst werden. **getInformationNeedModel** muss in **getModelName** geändert werden.
4. In der Methode `createPartControl` muss der Befehl InformationNeedModelForm c = **new** InformationNeedModelForm(parentEditor, parent, 0) angepasst werden. Für das neue Modell muss eine neue „Form“-Klasse angelegt werden. Wie das funktioniert, wird im Folgenden näher beschrieben.

„Form“-Klasse

Die „Form“-Klassen erben von der Klasse **Composite** aus dem Paket `org.eclipse.swt.widgets` und implementieren das Interface `IMQPForm` aus dem Paket `de.upb.pg.rmc.mqp.ui`. Jede „Form“-Klasse hat mindestens folgende Methoden:

- Konstruktor
- `initGUI`
- `createForm`
- `createBaseLayout`
- `getForm`
- `getMQPEditor`

Die Konstruktoren sehen alle wie folgt aus:

```
public Kontruktorname(IMQPEditor mqpEditor, Composite parent, int
  style) {
  super(parent, style);
  this.mqpEditor = mqpEditor;
  initGUI();
}
```

Listing 7.7: MQP: Form-Konstruktor

Als Parameter bekommt der Konstruktor eine Implementierung des Interfaces `IMQPEditor` übergeben. Dieses ist notwendig, um auf das Modell und die Elemente zugreifen zu können. Wie der Zugriff genau funktioniert, wird im Kapitel 7.2.3.3 beschrieben. Des Weiteren bekommt der Konstruktor ein Composite übergeben, in dem die Form angezeigt werden soll, sowie einen Style. Im Konstruktor wird der Konstruktor vom vererbten Composite und die Methode `initGUI` aufgerufen.

In `initGUI` werden die Elemente der **Form** initialisiert. Dafür werden zuerst die beiden Methoden `createBaseLayout` und `createForm` aufgerufen, in denen die Basiseigenschaften wie Überschrift und Größe gesetzt werden. Die **Form** selber enthält wiederum Sections. Die Sections dienen zur grafischen Trennung von inhaltlichen Themen. Sie sind zuklappbar, so dass der Benutzer sie ausblenden kann, wenn er in diesem Bereich nicht arbeitet. Wie die

Sections aufgebaut sind, wird später in diesem Teilkapitel noch erklärt. Ein Beispiel für das Benutzen einer Section sieht wie folgt aus:

```
Composite questionSection = new QuestionSection(this);
TableWrapData tdQuestionSection = new
  TableWrapData(TableWrapData.FILL_GRAB);
tdQuestionSection.colspan = 2;
questionSection.setLayoutData(tdQuestionSection);
```

Listing 7.8: MQP: Section

Dieses Beispiel zeigt einen Aufruf der **QuestionSection** aus dem **Informationsbedürfnismodell**. Dem Konstuktor der Section muss die Form übergeben werden, auf dem die Section angezeigt werden soll. Das geschieht hier über **this**. Es sollte einer Section auch immer ein **TableWrapData** fürs Layout zugewiesen werden. Über **colspan** legt man fest, dass die Section sich über zwei Spalten erstrecken soll, da die **Form** in diesem Beispiel, aus dem die Section entnommen wurde, auch in zwei Spalten geteilt wurde. Würde man **colspan** auf eins setzten, könnte man noch eine weitere Section daneben setzten. Bei zwei würde eine weitere Section unterhalb der Question Section angezeigt werden. Über setLayoutData wird das Layout der Section zugewiesen.

Die Methode `getForm` liefert die Form selber zurück.

Die Methode `getMQPEditor` liefert die Implementierung des `IMQPEditor` zurück. Über diese Implementierung kommt man an das Modell und die darin gespeicherten Elemente, um zum Beispiel weitere Elemente im Modell zu speichern oder darauf zu zugreifen. Dazu später im Kapitel 7.2.3.3 mehr.

Um eine eigene `Form`-Klasse zu schreiben, um zum Beispiel einen neuen Reiter für ein neues Modell zu erstellen, ist es am einfachsten, eine vorhandene Java-Datei zu kopieren und den Namen und die Methoden `initGUI` anzupassen. D.h., dass der Name der Klasse und des Konstruktors geändert werden muss und in der Methode `initGUI` andere oder neue Sections aufgerufen werden müssen. Die restlichen Methoden sollten nicht verändert werden. Wie eine Section aufgebaut ist, und wie man eine neue schreibt, folgt im nächsten Abschnitt.

„Section“-Klasse

Die „Section“-Klassen (Name + `Section.java`, z. B.: QuestionSection) erben von der Klasse `MQPBaseSection` aus dem Paket `de.upb.pg.rmc .mqp.ui`. Jede „Section“-Klasse hat mindestens folgende Methoden:

- Konstruktor
- `initGUI`

Eine Klasse `MQPBaseSection` wurde eingeführt, damit eine Verbindung zur Form, in der die Section liegt, hergestellt werden konnte. Dies ist über Methode `getMqpForm` möglich. Über die Form kann der Entwickler dann auf das Modell über die Methode `getMQPEditor` zugreifen. Ist keine Anbindung an das Modell notwendig, muss die Section nicht von **MQPBaseSection** erben.

Im Konstruktor wird nur der Konstruktor der `MQPBaseSection` aufgerufen. Der Konstruktor der `MQPBaseSection` ruft die Methode `initGUI` auf. Erbt man nicht von MQPBaseSection, muss man die Methode selber im Konstruktor aufrufen.

In der Methode `initGUI` werden die grafischen Elemente wie Buttons und Labels und dessen Layout angelegt.

Um eine eigene Section zu schreiben, sind folgende Schritte notwendig:

1. Anlegen einer neuer Klasse in `de.upb.pg.rmc.mqp.ui`. Dem Namen sollte ein „Section“ angehängt werden.
2. Die Klasse sollte von **MQPBaseSection** erben.
3. Der Konstruktor sollte den Konstruktor von **MQPBaseSection** aufrufen oder die Methode `initGUI`.
4. Die Klasse sollte eine Methode `initGUI` mit einer Annotation **@Override** enthalten.

Besonderheiten des Qualitätsmodells

Das Qualitätsmodell beinhaltet eine Besonderheit im Gegensatz zu den anderen Modellen. Im oberen Bereich des Modells befindet sich eine Canvas. Auf ihr können Graphen aus Qualitätsattributen und Qualitätscharakteristiken gezeichnet werden. Da die Darstellungen von Graphen mit herkömmlichen Formelelementen schwer realisierbar ist, wurde hier das GMF-Framework [GMF] verwendet. GMF wurde schon zum Erstellen des Klassen- und Zustandsdiagrammeditors verwendet. Wie die Zusammenführung von EMF- und GMF-Editoren in Eclipse funktioniert, wird im Artikel [WS08] beschrieben.

Besonderheiten des Präsentationsmodells

Im Präsentationsmodell wird zur grafischen Darstellung der Eingaben der vorherigen Modelle und der Beziehungen untereinander ein spezielles Toolkit prefuse visualization toolkit[14] verwendet. Von diesem Toolkit wurde die graphview[15] benutzt. Die **graphview** stellt die Elemente in einem Graphen dar. Um die Elemente aus den Modellen und die Messergbnise an den Graphen übergeben zu können, wurde ein Parser entwickelt, der die Informationen aus dem MQP-Modell ausliest. Der Parser befindet sich im Paket `de.upb.pg.rmc.mqp.ui` in der Datei **PMParser.java**. Der Parser liest das Ecore-Modell aus und schreibt die Knoten und Kanten in ein **byteArray** und übergibt dieses dem Graphen. Um die Messergebnisse aus dem Messmodell anzuzeigen, wurde eine zusätzliche **Propertiesview** entwickelt.

[14] http://prefuse.org/
[15] http://www.prefuse.org/gallery/graphview/

Die Repräsentation der Ergebnisse wäre zu unübersichtlich geworden, wenn die Ergebnisse in einem Graphen angezeigt würden. Für die **Propertiesview** gibt es in dem Paket `de.upb.pg.rmc.mqp.ui` die Klasse `PMProperties`. Wird ein Knoten eines Messergebnisses ausgewählt, werden dazu die Ergebnisse in der **Propertiesview** angezeigt. Dafür ruft die **Propertiesview** die passende Methode im Parser auf, zum Beispiel, wenn eine Basismessung ausgewählt wird, wird die Methode `ReadMeasureresultsBasemeasure` aufgerufen, welche die Ergebnisse an die **Propertiesview** zurückliefert.

7.2.3.3 Zugriff auf das Modell

In diesem Teilkapitel wird beschrieben, wie man Lese- oder Schreibzugriff auf das Modell mittels Java erlangen kann. Im Quellcode des MQP-Editors wird häufig auf das Modell zugegriffen. Diese Zugriffe sind notwendig, wenn der MQP-Editor zum Beispiel um weitere Modelle oder Elemente erweitert werden soll.

Lesezugriff auf das Modell

Möchte man auf die Objekte im Modell zugreifen, benötigt man dafür das Objekt **ModelQualityPlan**. Über dieses Objekt hat man Zugriff auf alle im Modell gespeicherten Objekte. An den **ModelQualityPlan** gelangt man über das implementierte Interface **IMQPSource**, welches eine Methode **getModelQualityPlan** besitzt, die ein Objekt **ModelQualityPlan** zurückliefert. Über die Methoden

- `getContextModel`
- `getInformationNeedModel`
- `getQualityModel`
- `getMeasurementModel`
- `getPresentationModel`

kann man auf die Modelle zugreifen. Die Modelle haben wiederum Methoden, mit denen man auf die Elemente im Model zugreifen kann. Zum Beispiel hat das Kontextmodell eine Methode `getPerson`. Diese Methode liefert Objekte aller Personen aus dem Modell in einer **EList** zurück.

```
ListIterator<OCLExpression> i =
  mqp.getMeasurementModel().getAuxiliaryFunctions().getOclExpression()
   .listIterator();
```

Listing 7.9: MQP: Lesezugriff auf das Modell

Das Beispiel zeigt den Zugriff auf das Messmodell und die darin vordefinierten OCL-Ausdrücke. Das Beispiel wurde dem **EditMeasureDialog** entnommen. Dieser Dialog ermöglicht das Anlegen und Editieren neuer Messungen. Dabei werden die OCL-Ausdrücke in einer Liste dem Benutzer zur Verfügung gestellt.

Schreibzugriff auf das Modell

Dialoge und Formulare dürfen nicht direkt auf dem Modell arbeiten, daher wurde das Model-View-ViewModel-Pattern [MVV] verwendet. Da keine **set**-Methoden verwendet werden können, um Objekte im Modell zu speichern, werden Commands[16] benutzt. Deren Verwendung hat zusätzlich noch den Vorteil, dass man sich bei der Entwicklung nicht um die **Undo**- und **Redo**-Funktionalität kümmern muss. Dieses geschieht automatisch bei Commands. Die Commands befinden sich im Packet `de.upb.pg.rmc.mqp.ui.commands`.

```
public class CreateMQPAuthorCommand extends ChangeCommand {

  private final ModelQualityPlan mqp;
  private final PersonData data;

  public CreateMQPAuthorCommand(ModelQualityPlan mqp, PersonData data)
    {
    super(new ChangeRecorder(), mqp);
    this.mqp = mqp;
    this.data = data;

    setLabel("add mqp author");
  }

  @Override
  protected void doExecute() {
    if (data.getSelectedPerson() == null) {
      Person person = MQPFactory.eINSTANCE.createPerson();
      person.setContextModel(mqp.getContextModel());

      person.setForename(data.getForename());
      person.setSurname(data.getSurname());
      person.setOffice(data.getOffice());
      person.setPhone(data.getPhone());

      MQPAuthor author = MQPFactory.eINSTANCE.createMQPAuthor();
      author.setPerson(person);
      author.setModelqualityplan(mqp);
      person.setMQPAuthor(author);
      mqp.getAuthor().add(author);

    }
    else {
      MQPAuthor author = MQPFactory.eINSTANCE.createMQPAuthor();
      author.setPerson(data.getSelectedPerson());
      author.setModelqualityplan(mqp);
      data.getSelectedPerson().setMQPAuthor(author);
      mqp.getAuthor().add(author);
    }
```

[16] http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/edit/command/package-summary.html

```
    }
}
```

Listing 7.10: MQP: CreateMQPAuthorCommand

Listing 7.10 zeigt ein Beispiel für ein Command. Der `CreateMQPAuthorCommand` wird verwendet, um Autoren für den MQP anzulegen. Die Klasse `CreateMQPAuthorCommand` erbt von der Klasse `ChangeCommand`. Sie wird verwendet, um Änderungen am Modell vorzunehmen. Durch ihre Verwendung ist es notwendig, die Methode `doExecute` zu überschreiben (@Override). Nur innerhalb der `doExecute`-Methode des Commands ist ein Schreibzugriff auf das Modell über **set**-Methoden erlaubt. Um einen Autor zu speichern, muss zuerst eine Person angelegt werden. Daher wird in der Methode als erstes ein Objekt Person erzeugt. Objekte des MQPs sollten immer über die `MQPFactory` erzeugt werden. Sie besitzt für jedes Element eine **create**-Methode. Beim Personen-Objekt können dann direkt die Eigenschaften wie Name, Vorname usw. gesetzt werden. Als nächstes wird über die `MQPFactory` ein Autor erzeugt, dem die Person zugewiesen wird. Anschließend muss der Autor dem Modell hinzugefügt werden. Dies geschieht über **mqp.getAuthor().add(author)**. Der Autor wird im Modell gespeichert, wenn das Command ausgeführt wird.

```
ChangeCommand cmd = new CreateMQPAuthorCommand(mqp, data);
getMqpForm().getMQPEditor().getEditingDomain().getCommandStack()
    .execute(cmd);
```

Listing 7.11: MQP: Verwendung CreateMQPAuthorCommand

Listing 7.11 zeigt wie das **CreateMQPAuthorCommand** verwendet wird. Das Beispiel ist der **AuthorsSection** entnommen. Über **new** wird ein neues Objekt des Commands erzeugt und anschließend über die Methode `execute` ausgeführt. Dem Konstruktor des Commands muss der MQP übergeben werden, damit der Command weiß, in welchem Modell der Autor gespeichert werden soll. Zusätzlich werden noch die Personendaten benötigt. Für die Personendaten gibt es eine eigene Klasse **PersonData**. Allgemein befinden sich die Klassen für die Daten, die den Commands im Konstruktor übergeben werden, im Packet `de.upb.pg.rmc.mqp.ui.viewmodel`.

7.2.4 Diagnose

Da die Diagnose dem Benutzer für gemessene Modelldefekte Diagnosen und dazu passende Therapien anzeigt, ist es evtl. notwendig neue Diagnosen und Therapien sowie Normalisierungen anzulegen. Dafür bietet die Diagnosekomponente mehrere Möglichkeiten. Zum einen kann die Diagnose um neue Normalisierungstypen erweitert werden. Zum anderen können innerhalb einer Konfigurationsdatei neue Symptome, Diagnosen, Therapien und Normalisierungen (passend zu neuen Normalisierungstypen) angelegt werden. Die Erweiterung der Diagnose um neue Symptome, Diagnosen und Therapien wird im folgenden Abschnitt (Abschnitt 7.2.4.1) betrachtet. Danach befasst sich Abschnitt 7.2.4.2 mit dem Anlegen einer neuen Normalisierung in der Konfigurationsdatei und Abschnitt 7.2.4.3 beschreibt das Erstellen eines völlig neuen Normalisierungstypen. Der letzte Abschnitt (Abschnitt 7.2.4.4)

gibt einen kurzen Überblick über denkbare Erweiterungen der Diagnosekomponente für die Zukunft.

7.2.4.1 Erweiterung der Konfigurationsdatei

Angenommen der Benutzer lässt im Modellqualitätsplan einen Indikator messen und möchte später mit Hilfe der Diagnose sein Modell auf evtl. auftretende Modellqualitätsmängel hin überprüfen. Dabei kann es vorkommen, dass ein vom Benutzer gewählter Indikator bzw. der dazugehörige Modelldefekt der Diagnose nicht bekannt ist und somit nicht angezeigt werden kann. Dies kann beispielsweise dadurch passieren, dass der Benutzer einen neuen Indikator im Messmodell angelegt hat, welcher der Diagnose nicht bekannt ist.

In diesem Fall muss die Konfigurationsdatei der Diagnose angepasst bzw. erweitert werden. Im folgenden Abschnitt wird anhand eines Beispiels, „ein benutzter Klassenname ist evtl. zu kurz und somit nicht aussagekräftig genug“, die Erweiterung um ein neues Symptom beschrieben. Hierzu sei angenommen, dass der Indikator durch den Benutzer bereits angelegt wurde und nun der Diagnose bekannt gemacht werden muss.

Wie bereits beschrieben, soll überprüft werden, ob die in einem Klassendiagramm erstellten Klassen, aussagekräftige Namen haben. Dies wird in diesem Beispiel dadurch überprüft, dass die Länge des Klassennamens bestimmt wird. Der dazugehörige Indikator ist mit dem Namen **Short class names** bezeichnet.

Begonnen wird damit, dass ein zum Indikator passendes Symptom erstellt werden muss. Dieses Symptom sieht wie in Label 7.12 abgebildet folgendermaßen aus:

```
<symptom name="Short class names (Symptom)" normalization="Fuzzy">
    <indicator name="Short class names" />
</symptom>
```

Listing 7.12: Beispiel: Anlegen eines neuen Symptomes

Ein Symptom besteht aus einem Namen, einer zugehörigen Normalisierung[17], und einem Indikatornamen. Der Indikatorname muss eindeutig sein und heißt in diesem Beispiel **Short class names**, wie er auch im Modellqualitätsplan bzw. genauer im Messmodell heißen muss.

Nachdem dem Indikator **Short class names** dem Symptom mit dem Namen **Short class names (Symptom)** zugeordnet wurde, muss diesem Symptom eine passende Diagnose zugewiesen werden. Eine Diagnose besteht dabei aus einem Namen, einer Beschreibung, die in der Diagnose-GUI die Diagnose beschreibt, und einer Referenz auf das eben erstellte Symptom. Listing 7.13 zeigt die fertige Eingabe. Zu beachten ist auch hier, dass der Name der Referenz auf das Symptom und der Name des oben angelegten Symtomes gleich sein müssen. Im vorliegenden Beispiel **Short class names (Symptom)**.

[17] Eingabe derzeit nicht relevant aber notwendig. Ist für eventuelle spätere Erweiterungen vorgesehen

```
<diagnosis name="Short class names (Diagnose)" description="Missing
  description of Diagnosis Short class names (Diagnose)">
    <reference name="Short class names (Symptom)" />
</diagnosis>
```

Listing 7.13: Beispiel: Anlegen einer neuen Diagnose

Nachdem nun eine Diagnose erstellt wurde, können der Diagnose derzeit zwei Arten von Therapien zugewiesen werden. Zum einen eine einfache Texttherapie und zum anderen eine Refactoringtherapie. Als erstes soll nun näher auf das Erstellen einer Texttherapie eingegangen werden.

```
<therapy name="Short class names (Therapie)" type="text">
    <reference name="Short class names (Diagnose)" />
    <description description="Text Therapy: Give your classes longer
      names" />
</therapy>
```

Listing 7.14: Beispiel: Anlegen einer neuen Texttherapie

Eine Texttherapie besteht aus einem Namen, einem Typen, welcher in diesem Fall **text** ist, einer Referenz auf eine Diagnose (in diesem Beispiel **Short class names (Diagnose)**) und einer Beschreibung (siehe Listing 7.14).

Das Anlegen einer passenden Refactoringtherapie ist sehr ähnlich. Sie besteht aus einem Namen, einem Typen (in diesem Fall **refactoring**), dem Namen einer Eigenschaft (siehe Eintrag **PropertyIdentifierList** in Tabelle 4.7 auf Seite 84), einer Referenz auf eine Diagnose und einer Beschreibung für die Anzeige in der GUI (siehe Listing 7.15).

```
<therapy name="Refactoring Therapy" type="refactoring">
  <property name="CLASS_BADNAME" />
  <reference name="Short class names (Diagnose)" />
  <description description="Refactoring Therapy: Bad Class name using
    property 'CLASS_BADNAME'" />
</therapy>
```

Listing 7.15: Beispiel: Anlegen einer neuen Refactoringtherapie

Nachdem nun wie in den Listings 7.12 bis 7.15 die Daten eingegeben wurden, ist der Diagnose nun der Indikator **Short class names** bekannt und würde bei entsprechenden Ergebnissen zu Verfügung stehen und in der GUI der Diagnose mögliche Modelldefekte anzeigen. Wie eine für die Anzeige von Diagnosen und Therapien notwendige Normalisierung in der Konfigurationsdatei angelegt wird, beschreibt der folgende Abschnitt.

7.2.4.2 Anlegen einer neuen Normalisierung

Im vorherigen Abschnitt wurde anhand eines Beispiels gezeigt, wie die Konfigurationsdatei um neue Symptome, Diagnosen und Therapien erweitert werden kann. In diesem Abschnitt geht es darum, wie die Diagnose um eine Normalisierung erweitert wird.

Um eine neue Normalisierung nutzen zu können, muss der Benutzer soweit diese noch nicht implementiert ist, wie in Abschnitt 7.2.4.3 beschrieben, eine neue Normalisierungsklasse anlegen und ggf. vorhandene Klassen anpassen. Wurde dies getan, kann die Normalisierung in der Konfigurationsdatei eingetragen werden. Möchte man die Werte für eine vorhandene Normalisierung an seine eigenen Bedürfnisse anpassen, so kann dies wie beispielsweise in Listing 7.16 zu sehen, ist mit dem Verändern der values für a und b durchgeführt werden.

```
<normalization type="Ordinal1" name="ordinal">
    <parameter name="a" value="10" />
    <parameter name="b" value="1" />
</normalization>
```

Listing 7.16: Beispiel: Struktur einer Normalisierung

Eine Normalisierung besteht dabei immer aus einem Typen, einem Namen und einer Menge von Parametern[18], die jeweils einen Namen und einen Wert haben. Mit Hilfe des Namens werden vom MQP übergebene Messergebnisse bzw. deren zugehörige Indikatoren den entsprechenden Normalisierungen zugeordnet. Der Typ (**type**) ist für eine spätere Erweiterung vorgesehen und hat derzeit noch keine Funktion. Listing 7.16 zeigt wie eine Normalisierung hier mit dem Namen **ordinal** und den beiden Parametern a und b aussieht.

7.2.4.3 Anlegen eines neuen Normalisierungstypes

In Abbildung 6.29 in Kapitel 6.3.4 sind die derzeitig implementierten Normalisierungstypen dargestellt. Möchte man einen neuen Normalisierungstypen definieren, so muss eine neue Klasse angelegt werden, welche von der Klasse `Normalization` erbt. In der von Benutzer angelegten Klasse muss dann die Methode `getNormalizedValue` auf jeden Fall implementiert werden, welche als Rückgabewert einen double-Wert hat. Neben dem Erstellen einer neuen Normalisierungsklasse muss auch die Klasse `NormalizationManager` erweitert werden. Hier wird mittels einer **if-else**-Abfrage ermittelt, welche Normalisierung aufgerufen werden muss. Es muss also so etwas wie in Listing 7.17 hinzugefügt werden.

```
else if (type.toLowerCase().equals("name of normalization class")) {
  normalizations.put(name, new "name of normalization class"(params));
}
```

Listing 7.17: Bekanntmachen der neuen Normalisierung

[18] abhängig von der jeweiligen Normalisierung

Vorraussetzung dafür, dass die Normalisierung auch genutzt wird ist, dass im MQP auch eine passende Skala existiert. Gegebenenfalls muss diese neu angelegt (siehe Abschnitt 7.2.4.2), einem Indikator zugewiesen und auch bewertet bzw. in einer Messung verwendet wurde.

7.2.4.4 Mögliche Erweiterungen

In diesem Abschnitt soll es darum gehen, welche Erweiterungs- bzw. Veränderungsmöglichkeiten für die Diagnose von Interesse sein könnten. Zum einen ist dem Benutzer bereits beim Lesen der Abschnitte, welche die Diagnose beschreiben sicherlich aufgefallen, dass die Diagnose derzeit keinerlei Intelligenz besitzt. Indikatoren werden vorgegebenen Symptomen, Symptome vorgegebenen Diagnosen und Diagnosen vorgegebenen Therapien zugeordnet. Sobald ein Indikator der Diagnose übergeben wurde, der nicht verarbeitet werden kann, also der Diagnose nicht bekannt ist, wird dieser nicht weiter betrachtet. An dieser Stelle ist nun eine gewisse Intelligenz (**AI**) vorstellbar, die selbstständig Zuordnungen durchführt und mit neuen zur Übergabe unbekannten Indikatoren „intelligent“ verfährt und weiter verarbeitet. Eine weitere Möglichkeit ist auch die Normalisierung mit einer gewissen Intelligenz auszustatten, da hier derzeit einem Indikator „starr“ eine Normalisierungsart zugeteilt ist. Wie man sehen kann, ist der Erweiterung bzw. Verbesserung dieser Komponente alleine durch die gerade aufgezeigten Beispiele ohne weiteres möglich. Mit den Informationen, die in den einzelnen Abschnitten dieser Dokumentation zu finden sind, sollte es realisierbar sein, schnell einen Einstig in das gesamte Projekt zu erhalten und die Diagnose „zum Leben zu Erwecken“.

7.2.5 Refactoring

Die Refactoring-Komponente basiert auf einer Grammatik, die ab Seite 234 beschrieben wird. Wie man den Refactoring-Wizard um neue Arten von Entscheidungen ergänzt, kann man auf Seite 240 nachlesen. Um größere strukturelle Umbauten bewerkstelligen zu können, widmet sich Kapitel 7.2.5.2 auf Seite 238 der Abhängigkeiten des Refactorings in Bezug auf die verwendeten Sprachen und Modelle.

7.2.5.1 Aufbau der Grammatik

Die Grammatik wurde mit dem *Java Compiler Compiler* (JavaCC) [JAV] erstellt und besteht aus einer einzigen Datei. JavaCC nutzt Dateien mit der Endung „.jj“, in denen sowohl die Struktur der Grammatik als auch Java-Fragmente enthalten sind.

JavaCC erzeugt aus der Grammatik-Datei mehrere Dateien mit Java-Quellcode, die die spezifizierte Grammatik parsen können. Um die Generierung zu starten, wird JavaCC mit dem Dateinamen als Parameter aufgerufen.

Die von der Grammatik definierte Sprache kommt zum Einsatz, wenn die Konfigurationsdatei geparst wird. Die Konfigurationsdatei wird in dieser Sprache formuliert, und während

des Parsens wird der dynamische Teil der Objekthierarchie der Refactoringkomponente aufgebaut. Diese Klassen, die während des Parsens erzeugt werden, sind in den Diagrammen im Kapitel 6.3.5.1 mit dem Stereotypen **parsed** gekennzeichnet.

Generell wurden einfache, kurze Codestücke direkt in die Grammatik geschrieben. Dagegen liegt die komplexe Logik, die über das Initialisieren der Objekthierarchie hinaus geht, in den Java-Klassen unter `de.upb.pg.rmc.refactoring`. Die Grammatik selbst liegt im src-Ordner[19] unter `refactoring/trunk/grammar` als `rmcrefactoring.jj`.

> **Hinweis:** Weiterführende Informationen zur generellen Funktionweise von JavaCC finden Sie auch in der Dokumentation[20] zu JavaCC.

Die Grammatik-Datei besteht aus folgenden Teilen:

1. Einem kurzen Block von Optionen, die das grundlegende Verhalten von JavaCC steuern, wie z.B. das Encoding der Datei selbst. Die Optionen stehen in einem Block, wie er in Listing 7.18 zu sehen ist.

```
options {
  ...
}
```

Listing 7.18: Grammatik-Abschnitt: Optionen von JavaCC

2. Die Hauptklasse `RefactoringParser` wird aus der Grammatik generiert. Diese Klasse führt das Parsen aus Java heraus durch. Hier findet man nur reinen Javacode inklusive Import-Anweisungen und der eigentlichen Klassendefinition (siehe Listing 7.19). Dieser lässt sich in zwei Bereiche aufteilen: Erstens wird der Parser selbst als Klasse definiert, inklusive der `parse`-Methode, die das Parsen anstößt. Zweitens sind hier mehrere kurze Methoden definiert, die im nächsten Teil der Grammatik genutzt werden.

```
PARSER_BEGIN(RefactoringParser)
...
PARSER_END(RefactoringParser)
```

Listing 7.19: Grammatik-Abschnitt: Definition der Klasse `RefactoringParser`

3. Nun beginnt die Definition der Inhalte der Grammatik (Listing 7.20). Es wird definiert, welche Zeichen als Whitespace anzusehen sind, und wie Kommentare formatiert sein müssen.

[19] Zu finden auf der Abgabe-CD.
[20] https://javacc.dev.java.net/doc/docindex.html

```
/* WHITE SPACE */

SKIP :
{
      " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

/* COMMENTS */

MORE :
{
      "//" : IN_SINGLE_LINE_COMMENT
    |
      "/*" : IN_MULTI_LINE_COMMENT
}
...
<IN_SINGLE_LINE_COMMENT, IN_FORMAL_COMMENT, IN_MULTI_LINE_COMMENT>
MORE :
{
    < ~[] >
}

/* RESERVED WORDS AND LITERALS */
```

Listing 7.20: Grammatik-Abschnitt: Whitespace

4. Es folgt die Definition von *Tokens* (Listing 7.21). Dies sind Schlüsselwörter, die als feste Zeichenkette einzelne Bereiche oder Elemente der Grammatik kennzeichnen. Ein Token besteht aus einem Tupel: Als Erstes wird ein eindeutiger String zur Identifikation des Tokens angegeben[21]. Als Zweites folgt die Zeichenfolge, die die Grammatik erkennen soll.

   Um ein neues Token hinzuzufügen, wird lediglich eine weitere Zeile hingefügt und damit eine ungenutzte ID gewählt.

```
TOKEN :
{
    < HEAD : "Refactoring grammar file of " >
  | < VER : " - Version 0.1">
  | < SREFAC : "SimpleRefactorings">
  ...
  | < SUBREFNAME : "SubRefactoringName">
  | < SUBCALC : "SubCalculation">
}
```

[21] Da diese Identifikation an keiner anderen Stelle in der Grammatik auftaucht, muss sie nur eine einzige Bedingung erfüllen: sie muss sich von allen anderen in der Tokenliste unterscheiden.

Listing 7.21: Grammatik-Abschnitt: Tokenliste

5. Neben den Tokens für Schlüsselwörter gibt es eine Reihe von Tokens, die Dinge definieren, wie z.B. Stringliterale. Hier wird beispielsweise festgelegt, ob und welche Zeichenketten mehrzeilig sein dürfen, und welche Zeichen als Buchstaben definiert sind. Listing 7.22 zeigt diesen Bereich.

```
/* LITERALS */

TOKEN :
{
    < STRING_LITERAL:
        "\""
      (    (~["\"","\\","\n","\r"])
           | ("\\"
                ( ["n","t","b","r","f","\\","'","\""]
                | ["0"-"7"] ( ["0"-"7"] )?
                  | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                  )
                )
      )*
        "\""
  >
...

TOKEN :
{
    < LB: "{" >
  | < RB: "}" >
  | < SEP: "," >
}

/* ELEMENTS OF THE GRAMMAR */
```

Listing 7.22: Grammatik-Abschnitt: interne Tokenliste

> **Warnung:** Änderungen in diesem Bereich wirken sich grundlegend auf weite Teile der Grammatik aus. Deshalb sollte man diese Art von Änderungen nur wohlbedacht durchführen.

6. Zuletzt folgen Terminale und Nichtterminale der Grammatik, die grob den einzelnen Klassen entsprechen. Hier sind daher Bereiche für Refactoringtypen, komplexe Refactorings, abstrakte Refactorings usw. zu finden. Die Bedeutung der einzelnen Elemente findet sich im Kapitel 4.2.5.2 auf Seite 82.

   Der Aufbau der Elemente wird anhand der Definition des Nichtterminals **Variable** aus Listing 7.23 erklärt. Die erste Zeile enthält den Rückgabetyp und den Namen des

Nichtterminals. Dann folgen zwei Blöcke[22]: Der erste nimmt Definitionen von Variablen auf, die im zweiten Teil genutzt werden. Dieser nimmt die eigentliche Definition auf, welche einerseits aus Zeichen besteht, die als Schlüsselwörter in der Sprache dienen (hier `"Variable"`, `"{"` und `"}"`). Zuweisungen wie `name = VariableName()` erfüllen den gleichen Zweck wie Variablenzuweisungen in Java, nur dass man den Variablen ein Terminal bzw. Nichtterminal zuweist. Innerhalb dieses Blocks steht in dem Block `{return Variable.create(type,name);}` der Rückgabewert von **Variable**.

Auf diese Art und Weise wurden alle Elemente der Sprache definiert. Eine detailliertere Beschreibung der Möglichkeiten von JavaCC findet sich unter der URL im Hinweis auf Seite 235.

```
Variable<?> Variable() :
{
  String name;
  Class<?> type;
}
{
  "Variable" "{" type = VariableType() name = VariableName()  "}"
    {return Variable.create(type,name);}
}
```

Listing 7.23: Grammatik-Abschnitt: interne Tokenliste

7.2.5.2 Abhängigkeiten der Grammatik

Wie man dem Kapitel 4.2.5.2 (Seite 82) entnehmen kann, hängt die Grammatik und damit der Inhalt der Konfigurationsdatei von der Abfragesprache ab, in der die Abfragen (**Queries**) ausgedrückt werden. Diese Sprache ist derzeit OCL.

Sie erlaubt außer reinen Abfragen auch das Ausführen von Java-Methoden, die im UML/Z-Modell definiert sind. Hierdurch werden die Queries flexibler, da Abfragen und Änderungen beliebig kombiniert werden können. Allerdings ergibt sich daraus auch, dass das UML/Z-Modell nicht losgelöst von der Grammatik verändert werden kann. Denn falls beim Ändern eine der Methoden betroffen sind, die aus OCL heraus genutzt werden können, ändert sich ebenfalls das Verhalten der Queries.
OCL erfüllt hier also eine Doppelrolle, die die Struktur der Grammatik vereinfacht: Es ist nicht relevant, ob eine Query lediglich eine Abfrage darstellt oder auch Änderungen durchführt.

> **Hinweis:** Näheres über die nutzbaren Methoden findet sich in Kapitel 8.2 auf Seite 243.

Falls die Abfragesprache geändert werden soll, betrifft dies indirekt zwar alle Teile, die Queries enthalten. Allerdings nutzen diese lediglich die Konstrukte, die direkt die Abfragen

[22] Hier ist ein Javablock gemeint, der in geschweifte Klammern eingefasst ist.

kapseln, und daher muss man bei diesen wenigen Konstrukten ansetzen, um die Anfragesprache zu ändern.

Die erste relevante Stelle zeigt Listing 7.24. Dort sehen man die Definition von **GlobalOcl**, welches Kurzschreibweisen aufnimmt, die in allen OCL-Queries verwendet werden können. Dies ist deshalb sinnvoll, da nicht-triviale Queries schnell unübersichtlich werden. Durch die Abkürzungen kann man die eigentlichen Queries besser lesen. **GlobalOcl** ist ohne direkten Bezug zu OCL implementiert, obwohl implizit davon ausgegangen wird, dass die Implementierung der Abfragesprache mit dieser Liste von Abkürzungen arbeiten kann.

```
void GlobalOcl(RefactoringManager manager) :
{Token string;}
{ "GlobalOCL" "{" string = <STRING_LITERAL_BESSER>
  {manager.setGlobalOCL(string.image.substring(1,string.image.length() -
   1)); } "}" }
```

Listing 7.24: Grammatik-Abschnitt: globale OCL-Statements

Eine Query selbst ist als **ContextQuery** definiert (Listing 7.25). Das Terminal verlangt zwei Parameter, die sich um Werte von Variablen kümmern:

IVariableHolder gibt die aktuellen Werte von Variablen zurück.

IVariableManager führt Zuweisungen an Variablen durch.

```
IContextQuery ContextQuery(IVariableHolder holder, IVariableManager
  manager) :
{
  String query;
  Variable<?> var = null;
  Collection<Variable<?>> env;
}
{
  "ContextQuery" "{" (var = Variable())? env = Parameter(holder) query
    = Query() "}" {return new ContextQueryImpl(var, env, query, holder,
    manager);}
}
```

Listing 7.25: Grammatik-Abschnitt: Query

Das Schlüsselwort **ContextQuery** nutzt die zwei Elemente **Query** und **Parameter**, die Listing B.4.2 zeigt. **Parameter** nimmt eine beliebige Anzahl von Variablen auf; genau diese Variablen können innerhalb der Query benutzt werden, und werden dann mittels des Arguments **IVariableHolder** durch ihre aktuellen Werte ersetzt. Das zweite Element **Query** definiert lediglich den Rahmen für die Zeichenfolge, die den Inhalt der Query aufnimmt.

```
Collection<Variable<?>> Parameter(IVariableHolder holder) :
{
  String name;
  Class<?> type;
  Collection<Variable<?>>  env = new HashSet<Variable<?>>();
```

```
}
{
  "Parameter" "{" (type = VariableType() ( name = VariableName()
    {env.add(Variable.create(type,name));})* ";" )*  "}"
  {
    return env;
  }
}

String Query() :
{String query;}
{
  "Query" query = StringLiteral() {return query;}
}
```

Listing 7.26: Grammatik-Abschnitt: Parameter der Query

So erhält eine **ContextQuery** alle notwendigen Java-Objekte und Parameter der Sprache, um während des Parsens als Java-Objekt erstellt werden zu können.

7.2.5.3 Decisions anlegen

Häufig ist eine Eingabe des Nutzers notwendig, um ein Refactoring durchführen zu können. Beispielsweise benötigt das Refactoring „Element umbenennen“ den neuen Namen des Elements. Eine solche Entscheidung, die der Nutzer mittels der GUI trifft, wird als *Decision* bezeichnet. Jeder Refactoringtyp enthält alle Decisions, die zu treffen sind. Diese Liste von Decisions wird dem Benutzer angezeigt, *bevor* das Refactoring durchgeführt wird. Daher gibt es ab dem Beginn des Refactorings keine unentschiedenen Decisions mehr und das Refactoring kann ohne Benutzerinteraktion ausgeführt werden.

Die notwendigen Entscheidungen werden dem Nutzer gemäß Kapitel 4.2.5.1 auf Seite 80 dargestellt. Für jede der drei dort erwähnten Decisiontypen gibt es eine GUI-Klasse, die dort als eine Seite des Wizards integriert wird. Tabelle 7.3 zeigt die derzeit vorhandenen Klassen, die jeweils eine der drei Decisions anzeigen.

Decision-Klasse	Beschreibung	GUI-Klasse
`StringInputDecision`	Eingabe eines Strings	`StringInputPage`
`ListQueryDecision`	Auswahl von Elementen aus einer Liste	`ListQueryDecisionPage`
`PredicateSplitting-Decision`	Teilt ein Prädikat in zwei Teile auf	`PredicateSplitting-Decisi onPage`

Tabelle 7.3: Refactoring: Implementierte Decisiontypen

Um eine neue Decision zu ergänzen, sind zunächst die folgenden Schritte notwendig:

1. Der Nutzer legt eine Klasse im Paket `de.upb.pg.rmc.refactoring` an, die alle Daten dieser Decision aufnimmt und von `Decision` im gleichen Paket erbt.

Diese Klasse benötigt zwei generische Parameter: Den Typ der Eingabe und den der Ausgabe. Bei der `StringInputDecision` wird klar, warum diese Typen verschieden sein können: Das *eine* vorhandene Prädikat ist die Eingabe, und die Ausgabe besteht aus den *zwei* Teilen der Eingabe.

2. Weiter wird eine GUI-Klasse in `de.upb.pg.rmc.refactoring.gui` erstellt. Sie erbt von `RefactoringDecisionPage` und stellt sowohl die GUI an sich zur Verfügung als auch die Überprüfung, ob die aktuelle Eingabe von der Decision akzeptiert wird. Dementsprechend zeigt die GUI Hinweise an, an welcher Stelle der Nutzer die Eingabe überprüfen sollte.

   Am Beispiel der Klasse `StringInputPage` wird die Funktionsweise deutlich: Die Methode `createControl` wird zum Erstellen der Wizard-Seite aufgerufen, und zwar zu einem Zeitpunkt, zu dem noch keine Daten der Decision vorliegen. Erst in `updateData` werden die initialen Daten der Decision in die GUI übernommen.

3. Hiernach bindet man die GUI-Klasse in die Methode `addPages` im `RefactoringWizard` ein, welche die einzelnen Seiten des Wizards erstellt. Dort fügt der Anwender eine entsprechende Behandlung der neuen Decision ein, damit diese von der eben erstellten GUI-Klasse repräsentiert wird.

Nun kann die Grammatik um die neue Decision ergänzt werden. Dazu führt der Nutzer folgendes zur Konfigurationsdatei hinzu:

1. Der Name der **Decision** wird nun als Token in der Grammatik definiert, damit dieser auch als Token erkannt wird. Näheres beschreibt der Abschnitt über Tokens auf Seite 236.
2. Listing 7.27 zeigt die Definition des Nicht-Terminals **DecisionsTypeElement**, bei der jeder **Decisiontyp** aufgelistet ist. Anhand dieser Liste kann beim Parsen entschieden werden, ob nur korrekte **Decisiontypen** in der Konfigurationsdatei benutzt werden. Daher muss hier eine neue Zeile mit dem neuen Typen eingefügt werden.[23]

```
String DecisionTypeElement() :
{}
{
  "StringInputDecision" {return "StringInputDecision";}
  | "ListQueryDecision" {return "ListQueryDecision";}
  | "PredicateSplitDecision" {return "PredicateSplitDecision";}
}
```

Listing 7.27: Grammatik: Liste der Typen von Decisions

Nun kann man den Parser durch JavaCC generieren lassen und testweise eine Decision vom gerade erstellten Typ in die Konfigurationsdatei einfügen. Diese sollte ohne Fehler geparst werden und dabei intern ein Objekt der neu erstellten Decision-Klasse erzeugen. Wenn der Nutzer das Refactoring über das ⟨`Refactoring`⟩-Menü (wie in Kapitel 4.2.5.1 auf Seite 79 beschrieben) startet, sollte die vorhin erstellte GUI angezeigt werden.

[23] Das neue Terminal sollte der neuen Decision entsprechend benannt werden.

8 API-Dokumentation

„Was guckt dein Gesicht so komisch..?“

Thomas

Die API-Dokumentation befindet sich aufgrund ihres Umfangs auf der beiliegenden CD-ROM. Im Folgenden werden die einzelnen Bestandteile der API-Dokumentation kurz erläutert.

8.1 Javadoc

Bei der Javadoc-API-Dokumentation handelt sich um eine Dokumentation aller öffentlicher (**public**) und geschützter (**protected**) Methoden und Attribute entsprechend dem Sun-Standard für Javadoc-API-Dokumentationen im HTML-Format.

Sie umfasst den gesamten innerhalb der Projektgruppe erstellten Java-Code, d.h. den gesamten Code im Paket `de.upb.pg.rmc`.

8.2 UML/Z-Dokumentation für den Zugriff mit OCL

Für den Zugriff auf das Modell sind die im Ecore definierten Attribute und Operationen ausschlaggebend, da mit OCL nur auf diese zugegriffen werden kann. Auf Attribute kann nur lesend zugegriffen werden, so dass Änderungen an UML/Z-Modellen ausschließlich mit Hilfe von im Ecore definierten Operationen durchgeführt werden können, die diese Änderungen dann vornehmen.

Wie neue Operationen (und auch Attribute) zum Ecore hinzugefügt werden können, ist in Kapitel 7.2.1.3 beschrieben. Die bestehenden Atttribute und Methoden sind in diesem Teil der API-Dokumentation beschrieben.

9 Resümee

„Na super!“

Peter

„Großartig!“

Tobias

9.1 Entstehung und Ziele der Projektgruppe

Im April 2007 trafen sich die Mitglieder der Projektgruppen RefaSo (R) und ModelCockpit (MC) zum ersten Mal. Schon vor Beginn des ersten Treffens war klar, dass beide Projektgruppen zusammen gelegt werden sollten. Jeder der Teilnehmer hatte seine eigenen Vorstellungen des Ablaufes einer Projektgruppe, sei es nun aus Horrorgeschichten vorheriger Projektgruppen oder eigenen Erfahrungen aus bisherigen Projekten. Wir hatten unterschiedliche Ziele und unterschiedliche Vorstellungen davon, wie wir unsere Ziele erreichen sollten. Nach genau einem Jahr haben wir als eine Projektgruppe - RMC - unser Projekt beendet. Dieses Fazit fasst nun zusammen, was wir (die Projektgruppe RMC) in einem Jahr erreicht haben. Nach einer Zusammenfassung unserer Ergebnisse bietet das Fazit einen Ausblick auf mögliche Erweiterungen unserer Software. Ergänzend zum Fazit beschreiben wir im Anhang A, was in der Projektgruppe gut, und was weniger gut gelaufen ist.

Die Projektgruppe ModelCockpit hatte ursprünglich das Ziel, die Qualität von UML-Modellen zu ermitteln und zu interpretieren. Dazu sollte zunächst als theoretische Grundlage ein Modell aufgestellt werden, mit dem es möglich ist, Qualität zu beschreiben. Anschließend sollte dann eine Software entwickelt werden, die Qualitätsmodelle erstellen und auf UML-Modelle anwenden kann. Das Projekt wurde zu Beginn auf der Projektgruppen-Webseite folgendermaßen beschrieben:

> „Modellbasierte Vorgehensmodelle spielen für die Überführung der Softwareentwicklung in ein ingenieurmäßiges Vorgehen eine herausragende Rolle. In Folge dieses Trends stellen Modelle zentrale Entwicklungsartefakte dar. Um Projektrisiken beherrschen zu können, muss die Qualität dieser Modelle erfasst und interpretiert werden. Dafür muss Modellqualität beschrieben, gemessen und adressatengerecht aufbereitet werden. Die Projektgruppe Model Cockpit soll hierbei den Modellnutzer mit einem Werkzeug unterstützen."
>
> Homepage PG ModelCockpit[1]

In der Projektgruppe Refaso sollte ein Refactoring-Werkzeug für eine formale Modellierungssprache erstellt werden. Dieses Werkzeug sollte dabei sicher stellen, dass sich das modellierte Verhalten durch die Refactorings nicht verändert. Auf der Web-Seite der Projektgruppe RefaSo wurde das Projekt so vorgestellt:

> „Aufgabenstellung ist Entwurf, Konzeption und Implementierung eines graphischen Werkzeuges, das Refactorings auf Modellen der Spezifikationssprache durchführt. Dabei soll dem Aspekt des „Erhaltens der Funktionalität" besonderer Augenmerk geschenkt werden. Dies soll durch den Einsatz eines Verifikationswerkzeuges (eines sogenannten Modelcheckers) geprüft werden, ein Verfahren, das herkömmliche Refactoring-Werkzeuge für objekt-orientierte Programmiersprachen auf Grund der fehlenden Semantik der Programme nicht leisten können. Aufgabe der Projektgruppe wird es auch sein, geeignete Modelchecker auszusuchen, und entsprechend der gewünschten Eigenschaft „Erhaltung der Funktionalität" einzusetzen."
>
> Homepage PG RefaSo[2]

Noch vor Beginn der Projektgruppen erkannten unsere Betreuer, Hendrik Voigt und Thomas Ruhroth, dass sich beide Themen wunderbar ergänzen. Durch die Kombination der beiden Themen ergibt sich ein Kreislauf, wie in Abbildung 9.1 dargestellt: zunächst wird die Modellqualität gemessen und so Fehler im Modell erkannt. Zur Korrektur dieser Fehler können dann Refactorings durchgeführt werden. Anschließend wird die Qualität erneut gemessen, um zu überprüfen, ob und inwiefern die Refactorings die Qualität des Modells verbessern konnten. So beschlossen Hendrik und Thomas also, Ihre Projektgruppen zu einer gemeinsamen Gruppe zu verbinden. Ehrlicherweise müssen wir zugeben, dass die für zwei eigenständige Projektgruppen zu geringe Teilnehmerzahl möglicherweise auch zur Entstehung dieser Idee beigetragen hatte.

[1] http://wwwcs.uni-paderborn.de/cs/ag-engels/ModelCockpit.html
[2] http://www.cs.uni-paderborn.de/index.php?id=3694&L=0

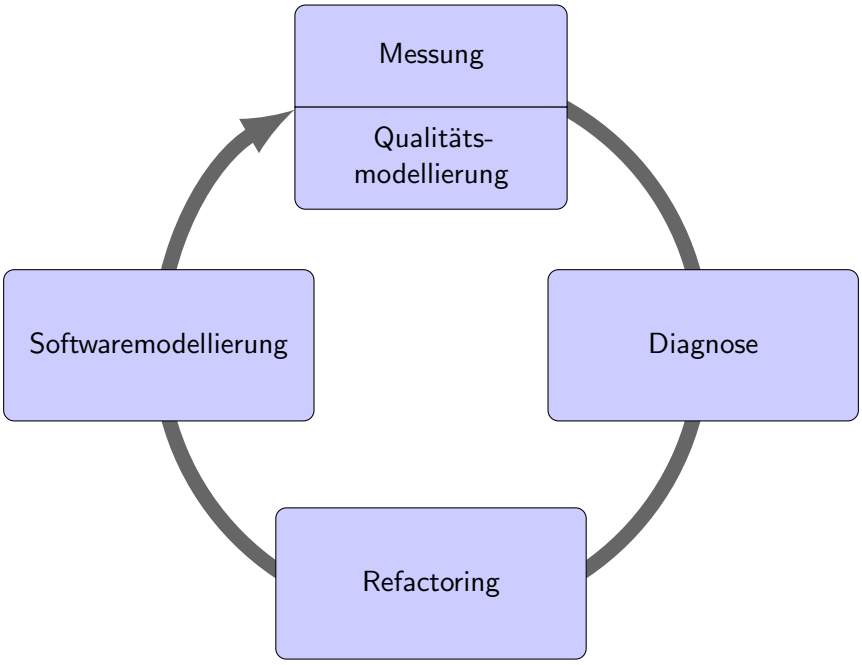


Abbildung 9.1: Qualitätskreislauf

Durch die Zusammenlegung der Projektgruppen ist ein Bindeglied nötig gewesen: die Qualität von Modellen sollte nicht einfach nur gemessen werden, es sollten auch automatisch Verbesserungsvorschläge erstellt werden. Diese Verbesserungsvorschläge sollten auch die Durchführung von Refactorings beinhalten können. Die Diagnose-Komponente ist die in der Projektgruppe realisierte Lösung dafür.

9.2 Vergleich von anderen Werkzeugen

Es gibt viele Werkzeuge zum Modellieren und Messen von Qualität und zur Durchführung von Refactorings. Bevor näher darauf eingegangen wird, was wir erreicht haben und wie, vergleicht dieser Abschnitt des Fazits zunächst unsere Zielsetzung mit den Möglichkeiten dieser bereits existierenden Werkzeuge und Methoden.

Syspect[3] ist ein Projekt der Universität Oldenburg. Syspect ist ein UML-Modellierungswerkzeug, und wie bei uns wurde auch dort die UML mit weiteren formalen Spezifikationssprachen kombiniert. Neben Object-Z [Smi00] unterstützt es CSP [Hoa04] und den Duration Calculus [CHR91], die in der Methode CSP-OZ-DC [HO02] zusammengefasst werden. Syspect-Modelle können mit einem Model-Checker verifiziert werden. Allerdings ist Syspect ein reines Modellierungswerkzeug, Refactorings werden nicht unterstützt, und die Modellierung von Qualität wird in diesem Projekt nicht betrachtet.

[3] http://syspect.informatik.uni-oldenburg.de/index.de.html

Together[4] ist ein kommerzielles UML-Modellierungswerkzeug der Firma Borland. Together wird hier stellvertretend für die meisten „typischen“ UML-Editoren betrachtet. Es unterstützt die Modellierung von UML 2.0. Zusätzlich können Geschäftsprozesse mit BPEL-4WS modelliert werden. Together kann verschiedene Qualitätsmetriken von Modellen messen. Auf Basis der OCL 2.0 [OMG06] können Metriken definiert und gemessen werden. Refactorings von Modellen werden nicht unterstützt.

Wir haben also nun gesehen, dass es bereits Werkzeuge gibt, die UML-Modellierung mit formalen Sprachen kombinieren oder Kennzahlen in Modellen messen können. Als nächstes betrachten wir, was unser Werkzeug zusätzlich leisten kann, und wie wir uns von anderen Werkzeugen abgrenzen.

9.3 Besonderheiten unserer Lösung

Unsere Software *RMC* versetzt den Anwender in die Lage, Qualität zu modellieren, zu messen und die Ergebnisse dieser Messung sofort automatisiert oder manuell mit Hilfe von Refactorings umzusetzen. Diese Funktionen haben wir in ein einziges Werkzeug integriert. Dadurch bekommt der Anwender die Möglichkeit seine Modelle in einem Qualitätskreislauf (Messung, Diagnose, Refactoring, erneute Messung) schrittweise zu verbessern. Durch diesen Qualitätskreislauf und die Integration der einzelnen Schritte in ein einziges Werkzeug hebt sich unsere Lösung von anderen Lösungen ab, die jeweils nur einzelne Aspekte wie z.B. Modellierung anbieten. Die einzelnen Teile unserer Software können auch unabhängig voneinander und unabhängig vom Qualitätskreislauf verwendet werden.

Unsere Lösung ist außerdem extrem flexibel, da sowohl Messungen als auch die Diagnose und die Refactorings durch Konfigurationsdateien definiert werden. Dadurch ist unsere Software ohne Änderungen am Programmcode erweiterbar.

9.3.1 UML-ähnliche Modellierung mit UML/Z

UML/Z besteht aus den aus der UML 2.0 bekannten Klassen- und Zustandsdiagrammen. Dabei wurden die Klassendiagramme um die Möglichkeit erweitert, das Verhalten von Operationen mit Z zu modellieren. Zustandsdiagramme in UML/Z beschreiben jeweils das Verhalten einer Klasse pro Zustandsdiagramm. Durch Z sind die Diagramme auch in ihrer Semantik klar und eindeutig spezifiziert.

Als Modellierungswerkzeug bietet RMC die grafische Modellierung von UML/Z-Klassendiagrammen und -Zustandsdiagrammen. RMC unterstützt dabei die „klassischen“ Funktionen eines UML-Modellierungswerkzeugs (Anlegen und Bearbeiten von Klassendiagrammen, Anlegen und Bearbeiten von Zustandsdiagrammen), erweitert um die Möglichkeit Z-Annotationen einzugeben.

Nach der Modellierung mit UML/Z kann der Anwender die Qualität der so erstellten Diagramme mit Hilfe von Modellqualitätsplänen messen.

Die Sprache UML/Z wird im Kapitel „Konzepte“ auf Seite 25 näher beschrieben.

[4] http://www.borland.com/us/products/together/index.html

9.3.2 Modellierung und Messung von Qualität

Mit unserer Software ist es möglich, Modellqualitätspläne einfach und intuitiv über eine grafische Benutzeroberfläche zu erstellen und somit Qualität zu modellieren. Über Modellqualitätspläne werden Qualitätsmaßstäbe definiert. Ein Modellqualitätsplan besteht aus fünf Teilmodellen:

- Kontextmodell (siehe Seite 163)
- Informationsbedürfnismodell Seite 163)
- Qualitätsmodell (siehe Seite 163)
- Messmodell (siehe Seite 166)
- Präsentationsmodell (siehe Seite 168)

RMC erlaubt die Eingabe der einzelnen Teilmodelle über eine grafische Benutzeroberfläche. Die so erstellten Modellqualitätspläne können anschließend auf verschiedene Modelle angewendet werden, um die Qualität dieser Modelle zu messen.

Neben einer ganzen Reihe vordefinierter Messungen besteht die Möglichkeit eigene Messungen auf Basis von OCL zu definieren. OCL-Abfragen werden verwendet, um bestimmte Eigenschaften einzelner Modellelemente zu messen, wie z.B. „Anzahl der Attribute einer Klasse“ oder „Länge des Namens einer Klasse“. Die Definition solcher Messungen erfolgt über eine grafische Benutzeroberfläche und ist somit einfach möglich, ohne dass Änderungen am Quellcode erforderlich sind.

Die Ergebnisse der Messungen werden dem Benutzer anschließend in Form eines interaktiven Graphen präsentiert. Sie dienen außerdem als Eingabe für die Diagnose, die aus den Ergebnissen weitere Maßnahmen ableiten kann.

9.3.3 Diagnose der Messergebnisse

Nachdem ein UML/Z-Modell mit Hilfe eines Modellqualitätsplanes gemessen wurde, kann auf Wunsch des Benutzers eine Diagnose dieser Ergebnisse durchgeführt werden. Dem Benutzer werden auf Basis dieser Diagnose verschiedene Verbesserungsvorschläge in Form von Diagnosen und dazugehörigen Therapien präsentiert. Die Diagnose ist vollständig konfigurierbar. Die Regeln, auf deren Grundlage die Diagnose erfolgt, werden dabei über eine XML-Konfigurationsdatei definiert.

Die Besonderheit unserer Software ist hier, dass die Diagnose bereits auf Modellebene erfolgt. Fehler bzw. Verbesserungsmöglichkeiten werden also schon während der Modellierung in den UML/Z-Diagrammen gefunden. Andere Werkzeuge[5] analysieren den Quellcode einer Software, nachdem dieser aus den Modellen erzeugt oder manuell geschrieben wurde. Fehler oder Verbesserungsmöglichkeiten, die erst in der Implementierung gefunden werden, sind wesentlich aufwendiger zu korrigieren. Bei uns erfolgt die Diagnose bereits während der Modellierung. Somit können viele Probleme bereits vor der Implementierung erkannt und beseitigt werden.

[5] http://www.uml.org/#Links-Tools

9.3.4 Konfigurierbare Refactorings

Refactorings [Fow99] sind Änderungen eines Programmes, die die Struktur des Programmes verbessern sollen ohne dabei das Verhalten des Programmes zu ändern. Klassischerweise werden Refactorings auf dem Quellcode eines Programmes durchgeführt. Unsere Software bietet Refactorings bereits auf Modellebene. Die Struktur der Software kann so schon während der Entwurfsphase verbessert werden. Refactorings können sowohl manuell ausgelöst werden als auch durch die Ergebnisse einer Messung mit einem Modelqualitätsplan.

Das Besondere an unserer Lösung ist dabei nicht nur die Durchführung der Refactorings auf Modellebene, sondern auch, dass die Refactorings nicht fest in unsere Software einprogrammiert sind. Stattdessen wurde ein Framework für Refactorings entwickelt, durch das neue Refactorings einfach über eine Konfigurationsdatei definiert werden können. Uns ist keine vergleichbare Software bekannt, in der Refactorings auf Modellebene so flexibel definiert werden können.

9.4 Vergleich des Ergebnisses mit der Zielsetzung

Wie in den meisten Softwareprojekten gibt es auch bei unserem keine hundertprozentige Übereinstimmung zwischen den vorher festgelegten Zielen und dem Ergebnis. Im Wesentlichen haben wir die uns gesetzten Ziele erreicht. In einigen Bereichen haben wir sogar deutlich mehr geliefert, als von uns verlangt wurde. So wird z.B. nirgendwo im Pflichtenheft gefordert, das Refactorings oder Diagnosen konfigurierbar sein müssen. Was die Flexibilität und Anpassbarkeit unserer Software betrifft, haben wir die Anforderungen also deutlich übertroffen.

Eine Anforderung, die leider nicht erfüllt wurde, sind konfigurierbare Qualitätsreports und alle damit verbundenen Anforderungen im Pflichtenheft. Es gibt lediglich eine Darstellung des Modellqualitätsplans mit allen Messergebnissen als interaktiven Graph.

Im Pflichtenheft wurde die Umsetzung von zehn Refactorings gefordert. Die zehn geforderten Refactorings wurden aus Zeitgründen nicht mehr vollständig implementiert. Dafür wurde aber ein sehr flexibeles erweiterbares Refactoring-Framework entwickelt, das wesentlich mehr bietet als im Pflichtenheft gefordert wurde. Durch diese Erweiterbarkeit bietet unsere Software einen größeren Mehrwert, als wenn wir zehn Refactorings erreicht, aber fest eingebaut hätten.

Unsere Software ist durch die Aufteilung in viele einzelne Plugins leicht wartbar und erweiterbar. Sollten sich später neue Entwickler in das Projekt einarbeiten müssen, können sie sich auf das für sie relevante Plugin konzentrieren, ohne dass sie sich mit den Implementierungsdetails der anderen Plugins beschäftigen müssen.

9.5 Ausblick

Dieses Fazit hat gezeigt, wie unsere Software die Qualität von Modellen messbar macht, wie sie über die eingebaute Diagnose Fehler und Verbesserungsmöglichkeiten findet, und wie sie

die Qualität des Modells anschließend durch Refactorings erhöhen kann. Zum Abschluss bietet das Fazit noch einen Ausblick darauf, welche Anforderungen an unsere Software nicht mehr umgesetzt werden konnten, und wie unsere Software weiter entwickelt werden könnte.

Teilweise wurden einzelne Anforderungen nicht so umgesetzt wie gefordert. Diese könnten durch nachfolgende Projektgruppen oder im Rahmen von Bachelor- und Masterarbeiten ergänzt werden. Außerdem ist RMC in vielen Bereichen von vornherein auf Erweiterbarkeit ausgelegt.

Für Modellqualitätspläne sollte es ursprünglich möglich sein Qualitätsreports zu definieren. Diese sollten verschiedenen Rollen (z.B. Professor oder Student) unterschiedliche Ansichten auf Messergebnisse zu einem Modell liefern. Für jeden Messwert sollte es z.B. möglich sein, unterschiedliche Darstellungsarten im Report auszuwählen (Text, Zahl, Balkendiagramm, Kreisdiagramm usw.). Aus Zeitgründen haben wir auf diese Funktionen verzichtet. Um Qualitätsreports nachträglich zu integrieren, wäre eine Erweiterung des MQP-Metamodells erforderlich: das Präsentationsmodell müsste im Metamodel angepasst und erweitert werden. Anschließend müsste die MQP-Benutzeroberfläche um Eingabe-, Bearbeitungs- und Darstellungsmöglichkeiten für Qualitätsreporte erweitert werden.

Die Refactoring-Komponente unterstützt nicht die ursprünglich geforderten 10 Refactorings. Da Refactorings aber über Konfigurationsdateien definiert werden, wäre es einfach möglich, weitere Refactorings zu definieren. Je nach Refactoring sind allerdings nicht alle für das Refactoring benötigten Operationen im UML/Z-Modell vorhanden, diese müssten dann nachträglich in das Modell integriert und implementiert werden. In der Entwicklerdokumentation wird diese Erweiterungsmöglichkeit ausführlich beschrieben.

In der Projektgruppe ModelCockpit gab es ursprünglich die Idee unsere Software in einer Vorlesung einzusetzen und so in der Praxis zu testen. Auf diesen Praxistest mussten wir aus Zeitgründen leider verzichten.

Neben den fehlenden Funktionen sind auch viele weitere Erweiterungen unserer Software denkbar.

Die Diagnose kann über eine Konfigurationsdatei um neue Symptome, Diagnosen und Therapien erweitert werden. Zusätzliche Normalisierungstypen können über Erweiterungen des Quellcodes ergänzt werden. Des weiteren könnte die Diagnose um zusätzliche „Intelligenz“ bei der Zuordnung von Indikatoren ergänzt werden. Erweiterungsmöglichkeiten für die Diagnose werden im Kapitel 7.2.4.4 ausführlich beschrieben.

In unserer Software wurden viele neue Konzepte wie die Modellierung von Qualität mit Modellqualitätsplänen oder die Bearbeitung von UML/Z-Modellen erstmals implementiert. Wir haben somit gezeigt, dass diese Konzepte auch implementierbar sind. Durch den modularen Aufbau unserer Software und die vielen Erweiterungsmöglichkeiten haben wir eine solide Basis geschaffen, die auch in Zukunft weiter genutzt und verbessert werden kann.

Literaturverzeichnis

[BeA03a] Aline Lúcia Baroni and Fernando Brito e Abreu. A Formal Library for Aiding Metrics Extraction. *4th International Workshop on Object-Oriented Re-Engineering at ECOOP 2003*, 2003. B.2.2, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.4.1, B.4.1, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2

[BeA03b] Aline Lúcia Baroni and Fernando Brito e Abreu. An OCL-Based Formalization of the MOOSE Metric Suite. *7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Darmstadt, Germany*, July 2003. B.3.1, B.3.1, B.4.1

[BPC05] Marcela Genero Bocco, Mario Piattini, and Coral Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4:59–92, 2005. B.3.1, B.3.2

[CHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. In *Information Processing Letters*, volume 40(5), pages 269–276, 1991. 9.2

[eA01] Fernando Brito e Abreu. Using OCL to Formalize Object-Oriented Metrics Definitions. Technical Report ES007/2001, FCT/UNL and INESC, Portugal, June 2001. Version 1.0. B.2.1, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.2, B.2.3, B.2.3, B.2.4, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.1, B.3.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2, B.4.2

[EMF] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/. 6.1, 6.3.1, 6.3.2, 6.3.3.2, 7.2.3.1, A.3

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999. 1.1, 9.3.4

[GMF] Graphical Modelling Framework. http://www.eclipse.org/gmf/. 6.1, 6.2.1, 6.3.2, 6.15, 6.3.2.1, 6.3.2.2, 7.2.3.2, A.3

[HL03] Erik Hatcher and Steve Loughran. *JAVA Development with Ant.* Manning Publications Co., Greenwich, CT, USA, 2003. 6.3.1

[HO02] Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining Specification Techniques for Processes, Data and Time. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 245–266, London, UK, 2002. Springer-Verlag. 9.2

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 2004. 9.2

[JAV] Java Compiler Compiler (JavaCC). https://javacc.dev.java.net/. 7.2.5.1, A.3

[MU05] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, pages 65–84, 2005. 6.1, 6.2.6, 6.3.7, 7.2.1.2

[MVV] Model-View-Controller (MVC) - reloaded as Model-View-View-Model (mvvm). http://www.codecomplete.de/blogs/xamlblog/archive/2007/09/20/mvc-model-view-controller-reloaded-as-mvvm-model-view-viewmodel.aspx. 7.2.3.3

[OMG06] OMG. Object Constraint Language. OMG Available Specification Version 2.0 formal/06-05-01, OMG, Mai 2006. 6.3.6, 9.2

[PRO] GMF Propsheet Customization. http://wiki.eclipse.org/GMF_Propsheet_Customization. 7.2.2.2

[RW07] Thomas Ruhroth and Heike Wehrheim. Refactoring Object-Oriented Specifications with Data and Processes. In *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, pages 236–251, 2007. 4.7, 6.3.5, 6.3.5.1, 6.3.5.1

[Smi00] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 9.2

[UML] UML-2-Tools. http://www.eclipse.org/uml2/. 6.3.1, 6.3.2, A.3

[VE08] Hendrik Voigt and Gregor Engels. Kontextsensitive Qualitätsplanung für Software-Modelle. *Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (eds.): Modellierung 2008, 12.-14. März 2008, Berlin*, GI-Edition, Lecture Notes in Informatics, LNI 127:165–180, 2008. 2.1, 6.3.3.2, 6.3.3.2, 6.3.3.2, 6.3.3.2

[WD96] Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement and Proof*. Prentice-Hall, 1996. 2.1, 6.2.6, 6.3.1.1, A.2

[Weh00] Heike Wehrheim. Specification of an Automatic Manufacturing System: A Case Study in Using Integrated Formal Methods. In *FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 334–348, London, UK, 2000. Springer-Verlag. 2.1

[WS08] Volker Wegert and Alex Shatalin. Integrating EMF and GMF Generated Editors. http://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html, 2008. 7.2.3.2

Index

Abgeleitete Messungen, 70
AfterTemplate, 182
API, 243
ApplicabilityCondition, 85, 182
Architektur, 143, 202
Architekturkomponenten, 153
Artefakt, 62
Autor, 61

Basismessungen, 70
BeforeTemplate, 182

Calculation, 182
CalculationTemplate, 182
Canvas, 45, 56
Commands, 229
ComplexRefactoring, 87
Comunity Z Tools, 193
CZT, 193
CZTWrapper-Komponente, 145, 193, 203

Decision, 86, 184, 240
DecisionTypeElement, 86
DecisionValidation, 86, 182
Diagnose, 76
Diagnose Skala, 174
Diagnostics-Komponente, 172, 203
Diagramm anlegen, 36

Eclipse Modelling Framework, 153, 158, 223
Ecore, 223
EMF, 153, 158, 223
EnforcedRefactoring, 88

Form, 225
Frage, 65
Fragegruppe, 66

Generierung, 210
Genmodel, 223
GMF, 158

Grammatik, 234
Graphical Modelling Framework, 158
Grenzwert, 76, 179

IModel-Schnittstelle, 147, 202
IMQP-Schnittstelle, 149, 202
IMQPInternalModel-Schnittstelle, 202
Informationsbedürfnismodell, 65
IOCL-Schnittstelle, 202
IOCLParser-Schnittstelle, 149
IRefactoringDiagnostics-Schnittstelle, 148, 202
IRefactoringEditor-Schnittstelle, 202
IZParser-Schnittstelle, 150, 202

JavaCC, 234
Javadoc, 243

Klassendiagramm, 40
Klassendiagrammeditor, 160
Komponenten, 35, 143, 202
Konfigurationsdatei, 60, 79
Konfigurationsdatei einlesen, 79
Kontextmodell, 61

Lesezugriff, 228
ListQueryDecision, 185

Messmodell, 70
Messung, 29
ModelCockpit, 23, 245
Modelldefekt, 134
ModelQualityPlan-Komponente, 145, 160, 203
Modelqualitätsplan, 58
Moderator, 61
MQP, 58, 223
MQP-Metamodell, 162

Normalisierung, 175, 178, 230, 233
Normalisierungstyp, 233

Object Constraint Language, 189
OCL, 189, 238
OCL-Parser, 189
OCLWrapper-Komponente, 145, 189, 203
Outline, 45

Palette, 42, 54
Pflichtenheft, 250
Precondition, 182
Projekt, 36
PropertyIdentifierList, 84
Präsentationsmodell, 73

Qualitätsattribut, 68
Qualitätscharakteristik, 68
Qualitätsmodell, 68
Qualitätsreport, 250

Refactoring, 30, 79, 85
Refactoring-GUI, 185
Refactoring-Komponente, 145, 180, 203
Refactoring-Schnittstelle, 147
Refactoringassistent, 80
RefactoringBase, 87
Refactoringkandidat, 174
Refactoringmenü, 79
Refactorings laden, 79
Refactoringtherapie, 76, 232
RefactoringType, 85
RefaSo, 23, 245
Reiter, 224
RMC-Komponenten, 143
Rolle, 62

Schnittstellen, 147, 202
Schreibzugriff, 229
Section, 226
SimpleRefactoring, 86
SplitPredicateDecision, 185
StringInputDecision, 185
StumblingBlock, 85, 184
SubRefactoring, 87
SubRefactoringCondition, 182

Textkandidat, 174
Texttherapie, 76, 172, 232
Therapiekandidat, 174
Token, 236

TypeIdentifierList, 84

UML/Z-Editor, 36
UML/Z-Editor-Komponente, 145, 158, 203
UML/Z-Model-Komponente, 145, 153, 203

Variablen, 185

Z-Editor, 48
Z-Parser, 193
Ziel, 66
Zustandsdiagramm, 53
Zustandsdiagrammeditor, 160

A Unsere Vorgehensweise

Dieser Abschnitt beschreibt die Organisation unserer Projektgruppe und unsere Vorgehensweise, die von uns dafür verwendete Software, und die Erfahrungen, die während der Projektgruppe gemacht haben. Wir möchten so unsere Erfahrungen und Tipps an zukünftige Projektgruppen weiter geben, die möglicherweise RMC weiter entwickeln werden.

A.1 Projektorganisation

Unsere Projektgruppe bestand aus elf Studenten und - wegen der Zusammenlegung von ursprünglich zwei Gruppen zu einer - aus zwei Betreuern. Innerhalb der Gruppe wurden verschiedene organisatorische Aufgaben verteilt, die jedes Mitglied zusätzlich zur normalen Projektgruppenarbeit übernehmen musste. Bei der Verteilung der Aufgaben haben wir darauf verzichtet, einen Projektleiter zu benennen. Alle Entscheidungen wurden demokratisch durch Mehrheitsbeschluss durchgeführt. Die Aufgabenbereiche waren:

- Code-Administration
  - Buildmanagement (Mathias)
  - Coding Conventions (Martin)
- IT-Administration (Tobias)
- Dokumentation (Rudolf)
- Integrator (Peter)
- Ressourcen-Management (Christoph)
- Qualitäts-Management (Steffen)
- Entwicklung (Bernhard)

Die Projektgruppe begann im April 2007 und dauerte bis April 2008. In dieser Zeit wurde das Projekt durchgehend bearbeitet, auch in der vorlesungsfreien Zeit. Es gab lediglich für einzelne Projektmitglieder Urlaub oder Klausurfrei. In dieser Zeit gab es jeden Donnerstag einen festen Termin, an dem sich die ganze Projektgruppe getroffen hat, um den aktuellen Status des Projekts zu besprechen und Entscheidungen zu treffen. Im Abstand von ca. sechs Wochen haben wir unsere Ergebnisse unseren Professoren in Form eines Lenkungskreises vorgestellt. Ein Lenkungskreis bestand aus drei Mitgliedern der Projektgruppe, unseren Betreuern und unseren Professoren.

Der Projektverlauf lässt sich grob in fünf Phasen unterteilen: Seminarphase, Entwurf, Implementierung, Test und Dokumentation.

In der Seminarphase hat jedes Projektgruppenmitglied ein für die Projektgruppe relevantes Thema aufgearbeitet und den anderen Projektgruppenmitgliedern in einer Präsentation sowie einer schriftlichen Ausarbeitung vorgestellt. Oft waren diese Präsentationen auch mit einer Entscheidungsfindung verbunden, z.B. die Auswahl eins Versionskontrollsystems oder Wahl zwischen Word und Latex für die Projektdokumentation.

In der Entwurfsphase wurde zunächst ein Pflichtenheft erstellt. Anschließend wurde ein Grob- und danach ein Feinentwurf für alle Teile unserer Software entwickelt.

Mit der Implementierung der Software haben wir im Oktober 2007 begonnen. Die Projektgruppe wurde in Teilgruppen für die einzelnen Komponenten aufgeteilt (Editoren, Refactoring, Diagnose, OCL, MQP), die ihre jeweiligen Komponenten relativ unabhängig voneinander entwickelt haben.

Für die Implementierung haben wir mehr Zeit benötigt, als wir vorher geschätzt hatten. Deshalb konnten wir erst später als geplant, nämlich im März 2008, mit der Testphase beginnen. Der Test der Software erfolgte hauptsächlich in einer Testwoche, bei der alle Projektgruppenmitglieder die ganze Woche im Poolraum der Projektgruppe gearbeitet haben, um Fehler möglichst sofort zu beheben. Während der Testwoche konnten wir über 150 Fehler finden und beheben.

Abgeschlossen wurde die Projektgruppe durch eine Dokumentationsphase von März 2008 bis April 2008, in der diese Abschlussdokumentation erstellt wurde.

A.2 Themen der Seminarphase

Software Qualitätsmanagement: Qualität ist kein eindeutig definierbarer Begriff, so dass es verschiedene Möglichkeiten zur Interpretation von Qualität gibt. Bei der Vorstellung des Themas wurden daher verschiedene Ansätze näher erläutert, u.a. Produkt- und Prozessqualität. Ein weiterer Punkt war eine kurze Einführung in die verschiedenen Testverfahren.

UML-Klassen- und Zustandsdiagramme und deren Beziehung: Für den Entwurf unseres Tools RMC waren unter anderem Grundlagen der UML unabdingbar, die daher in den Entwicklungsprozess eingeflossen sind. In dieser Seminararbeit wurde der Fokus auf die beiden Diagrammtypen „UML-Klassen- und Zustandsdiagramme“ gesetzt, um den Projektgruppenteilnehmern einen Überblick zu diesen beiden UML-Diagrammtypen zu geben. Die Aufgabe war dabei aufzuzeigen, was UML-Klassen- und Zustandsdiagramme sind und wie diese beiden Diagrammtypen miteinander in Verbindung stehen. Dabei wurde im Verlauf der Ausarbeitung auf die einzelnen Teile und Beziehungen der Diagramme eingegangen und ihre Symbolik aufgezeigt.

Qualitätsmodelle: Qualitätsmodelle definieren ein grundlegendes Qualitätsverständnis für die Bewertung von Modellen. Sie liefern eine Möglichkeit, die Qualität eines Modells präzise erfassen und messen zu können. Diese Qualitätsmodelle waren im Hinblick auf die Entwicklung von Modellqualitätsplänen ein wichtiger Einstiegspunkt in die Beurteilung der Qualität von Modellen.

Dokumentationsstandards: Diese Ausarbeitung beschäftigt sich zum Einen mit dem Thema Dokumentation - Warum? Was? Wie? Das heißt, warum sollte ich überhaupt etwas dokumentieren? Was genau soll ich dokumentieren? Und wie dokumentiere ich überhaupt? Das zweite große Kapitel sind die Werkzeuge zur Dokumentation, die für uns in der Projektgruppe interessant sein könnten. Dabei unterscheiden wir zwischen Dokumentationswerkzeugen, die für die „normalen“ Dokumente im Projektalltag helfen, Dokumente zu erstellen, und den Quellcodedokumentationswerkzeugen, die aus unseren Quellcodekommentaren automatisch eine Dokumentation generieren.

Software Configuration Management: Es wurden verschiedene Versionskontrollsysteme verglichen, um für die Projektgruppe ein geeignetes System auszuwählen. Nach dem Vortrag wurde Subversion als Versionskontrollsystem ausgewählt.

Z-Notation: Mit Z und Object-Z lassen sich präzise Umschreibungen von Systemen und ihre Komponenten mit Hilfe mathematischer Definitionen angeben. Die Ausarbeitung bezieht sich zum größten Teil auf das Buch von Jim Woodcock und Jim Davies Using Z [WD96]. Z ist ein Teil der Modellierungssprache UML/Z und insofern bedeutend für das Verständis von UML/Z und die Einsatzmöglichkeiten innerhalb von Modellstrukturen.

Smells und Refactoring: Smells sind Indikatoren für schlechte Strukturen von Quelltext. Daher kann man mittels Smells die Stellen im Code lokalisieren, die Mängel aufweisen. Diese können mittels Refactorings behoben werden.

Refactoring von statischer und dynamischer Sicht: Das Thema stellt die Konzept für die Refactorings auf der Spezifikation CSP-OZ dar. Es dient als Grundkonzept für die Refactorings auf UML/Z-Modellen.

OCL als Anfragesprache: OCL ist eine Spezifikationssprache, die von der Object Management Group (OMG) parallel zur Unified Modeling Language (UML) entwickelt wird. Zunächst als Hilfsmittel zur textuellen Spezifikation von Bedingungen wie Invarianten, Vor- und Nachbedingungen mit Hilfe von Constraints entwickelt, ist es darüber hinaus möglich, OCL als Anfragesprache zu nutzen und Eigenschaften von Modellen bzw. Modellelementen zu ermitteln.

Subjektive Messungen - Reviews: Ein wichtiger Faktor in der Softwareentwicklung ist die Qualität der implementierten Produkte und der zugrundeliegenden Entwürfe und Modelle. Je früher Fehler, Mängel und Schwachstellen entdeckt werden, umso einfacher und günstiger lassen sich diese Mängel beheben. Jedoch lassen sich nicht alle Fehler durch automatisiertes Überprüfen der Dokumente und des Quellcodes auffinden. Es sind zusätzliche Überprüfungen durch Menschen nötig. Mit diesem Gebiet beschäftigen sich die subjektiven Messung, die ein Teilgebiet der analytischen Qualitätsmaßnahmen darstellt.

Kennzahlen für Modelle: Um die Qualität eines Modells zu messen, ist es zunächst nötig, Kriterien festzulegen, die als Indikatoren für ein gutes oder schlechtes Modell angesehen werden können. Dies kann z.B. die syntaktische Korrektheit des Modells bezüglich seiner Modellierungssprache sein, ob das Modell in sich konsistent ist, ob das Modell vollständig bezüglich einer Spezifikation ist oder ob einzelne Teile des Modells untereinander vollständig sind.

A.3 Verwendete Werkzeuge

In der Projektgruppe haben wir nur Software verwendet, die entweder frei verfügbar, oder für Studenten kostenlos über die MSDN AA verfügbar ist:

Sun Java SDK 1.6: Von unseren Betreuern wurde uns vorgegeben, unsere Software auf Basis von Java zu entwickeln. Wir haben das aktuelle Java SDK 1.6 von Sun eingesetzt.

Eclipse 3.3.x: Als Entwicklungsumgebung hatten wir uns für Eclipse entschieden, da Eclipse eine sehr gute und kostenlose Java IDE ist, und wir Eclipse nicht nur als IDE, sondern auch als Plattform für unsere Software nutzen wollten.

Eclipse UML2 Tools: Unser UML-Editor basiert auf den Eclipse UML2 Tools [UML]. Alle nicht benötigten UML-Diagrammtypen wurden aus den UML2 Tools entfernt, anschließend wurden unsere Erweiterungen (Z) in das Metamodell und die Editoren integriert.

Eclipse UML2 Model Development Tools: Eclipse UML2[1] ist eine Implementierung des UML2-Metamodells. Wir haben diese Implementierung als Grundlage für das UML/Z-Metamodell verwendet.

Eclipse Modeling Framework (EMF): Das Eclipse Modeling Framework [EMF] ist ein Framework zur Erzeugung von Modellen. Wir haben EMF für das UML/Z-Metamodell und das MQP-Metamodell verwendet. Aus den EMF-Modellen wird der Code unserer Modelle generiert.

Graphical Modeling Framework (GMF): Um grafische Editoren für EMF-Modelle zu erhalten gibt es das Eclipse Graphical Modeling Framework [GMF]. Wir verwenden es für die UML/Z-Editoren sowie den grafischen Editor im Qualitätsmodell des Modellqualitätsplans.

LaTeX: Zum Erstellen von Dokumenten wie Architektur, Pflichtenheft, Abschlussdokumentation usw. haben wir LaTeX verwendet.

Subversion, Subclipse, TortoiseSVN: Als Versionskontrollsystem haben wir Subversion eingesetzt. Als Clients wurden unter anderem das Eclipse Plugin Subclipse und der Windows Client TortoiseSVN verwendet.

Microsoft Visio und Borland Together: Für UML-Diagramme haben wir zunächst Visio 2007 und Visio 2003 verwendet. Später haben wir dann Borland Together eingesetzt, da Together Java-Code aus Diagrammen generieren und Diagramme aus bereits vorhandenen Code erstellen kann.

JavaCC: Ein Teil des Quellcodes des Refactorings wird über JavaCC [JAV], den Java Compiler Compiler generiert.

JavaDoc: Für die Generierung unserer API-Dokumentation haben wir JavaDoc verwendet.

Apache Ant: Während ein Teil der Projektgruppe ausschließlich mit Eclipse gearbeitet hat, um die Projekte zu kompilieren, wurden in anderen Teilgruppen umfangreiche Ant-Skripte eingesetzt. Unter anderem wurden JavaCC und JavaDoc über das Ant-Skript aufgerufen.

[1] http://www.eclipse.org/modeling/mdt/downloads/?project=uml2

Bugzilla: Gegen Ende der Projektgruppe und insbesondere in der Testphase haben wir Bugzilla verwendet, um alle gefundenen Fehler zu protokollieren und an einzelnen Entwicklern zuzuweisen.

JUnit: Für einige Komponenten wurden Unit-Tests erstellt, dafür haben wir JUnit verwendet.

Microsoft Project 2003: Unser Projektplan wurde mit Microsoft Project erstellt.

eGroupware: Zur Planung von Terminen, Urlaub usw. wurde die Software eGroupware eingesetzt.

Mediawiki: Für den Austausch von verschiedenen Informationen haben wir ein Wiki eingesetzt.

A.4 Hinweise für zukünftige Projektgruppen

Diese Dokumentation möchten wir auch nutzen, um unsere Erfahrungen an zukünftige Projektgruppen weiter zu geben. Deshalb stellen wir in diesem Abschnitt einige Punkte vor, die gut gelaufen sind, sowie Dinge, die wir beim nächsten Mal anders machen würden.

A.4.1 Zeitmanagement

Bei unseren wöchentlichen Gruppentreffen wurden jedem Mitglied der Projektgruppe Aufgaben zugewiesen und in einem mit Microsoft Project erstellten Projektplan festgehalten. Die Planung erfolgte dabei im Detail für die nächsten 1-2 Wochen und grob für die nächsten 2-3 Monate. Um die Dauer der einzelnen Aufgaben zu bestimmen, wurde die bis zu einer Deadline verfügbare Zeit auf die einzelnen Aufgaben verteilt.

Leider wurden die vorgegebenen Zeiten in den meisten Fällen nicht eingehalten und Deadlines immer wieder verschoben. Möglicherweise wäre es sinnvoller gewesen, zuerst den Aufwand für die zu erledigenden Aufgaben zu schätzen und daraus dann einen Zeitplan zu erstellen. Auf nicht erreichbare Lenkungskreisziele oder feste Deadlines, die dadurch überschritten worden wären, hätte man dann z.B. durch Streichung von Funktionen mit niedriger Priorität reagieren können. In diesem Punkt gab es unterschiedliche Meinungen in der Projektgruppe, so dass das Verfahren trotz der selten eingehaltenen Ziele bis zum Ende der Projektgruppe beibehalten wurde.

A.4.2 Kein Projektleiter

Wir hatten bei der Aufteilung der Rollen auf einen Projektleiter verzichtet. Dies hatte allerdings den Nachteil, dass für viele Entscheidungen sehr lange Diskussionen erforderlich waren, bis die Gruppe gemeinsam eine Entscheidung treffen konnte. Manchmal war es auch nicht möglich, sich auf eine Entscheidung zu einigen - ein Projektleiter hätte in so einem Fall eine Entscheidung treffen können.

A.4.3 Zusammenarbeit vor Ort

Vor allem ab ca. der Mitte der Projektgruppe haben wir verstärkt vor Ort, also im Projektgruppenpoolraum, zusammen gearbeitet. Dadurch waren wir oft produktiver, als wenn jeder für sich allein zu Hause gearbeitet hat. Etwas problematisch war dabei allerdings gemeinsame Arbeitszeiten zu finden, da einige Projektgruppenmitglieder bevorzugt nachts, andere morgens gearbeitet haben. Das hat teilweise zu Problemen geführt, wenn z.B. machts Änderungen durchgeführt wurden, durch die morgens dann Fragen und Probleme entstanden sind.

A.4.4 Entscheidungsfindung mit Hilfe eines Kriterienkatalogs

Am Anfang der Projektgruppe kam es bei Entscheidungsfindungen oft zu endlosen Diskussionen, die sich immer wieder im Kreis bewegt haben, ohne erkennbare Fortschritte zu erzielen. Später haben wir dann die möglichen Alternativen systematisch verglichen, indem wir die Vor- und Nachteile in einer Tabelle aufgelistet und gewichtet verglichen haben. Dadurch konnten wir Entscheidungen schneller und besser nachvollziehbar treffen.

A.4.5 Zwei Betreuer

Unsere Projektgruppe wurde von zwei Betreuern unterstützt, die teilweise sehr unterschiedliche Arbeitsweisen und Ansichten hatten. Beide kommen aus unterschiedlichen Arbeitsgruppen und konnten die Projektgruppe jeweils mit Fachwissen aus ihrem Arbeitsgebiet unterstützen. Hendrik Voigt hat hauptsächlich den Bereich Qualitätsmodellierung unterstützt, Thomas Ruhroth die Bereiche UML/Z, Refactoring und Diagnose. Auch wenn sich unsere Betreuer nicht immer sofort einigen konnten, hat sich das ingesammt doch eher positiv auf unser Ergebnis ausgewirkt. Durch die „doppelte Betreuung“ war bei Fragen und Problemen fast immer einer der Beiden als Ansprechpartner erreichbar.

A.4.6 Spezialisten für einzelne Themen vermeiden

Leider gab es einige Bereiche und Aufgaben, mit denen nur einzelne „Spezialisten“ vertraut waren. Dies hat zu Problemen geführt, wenn es Rückfragen in diesen Bereichen gab, und die Spezialisten gerade nicht erreichbar waren. Hier hätten wir darauf achten sollen, dass sich mit allen Aufgaben und Themen immer mindestens zwei oder mehr Personen auskennen.

A.4.7 Freizeitaktivitäten

Neben unserer Projektgruppenarbeit haben wir einige gemeinsame Veranstaltungen organisiert, z.B. eine Kanu-Tour, Grillen und einen Bowling-Abend. Diese Veranstaltungen haben sich sehr positiv auf die Teambildung ausgewirkt.

B Vordefinierte Messungen mit OCL

B.1 Scale of measure results

A scale is a instance of a `ScaleType`. The `ScaleTypes` are found in the meta measurement model of the MQP in the section 6.3.3.2.

Name:	Schulnoten als Zahlen
TypeOfScale:	Integer
UpperBound:	6
LowerBound:	1
Interval:	1

Name:	Integer
TypeOfScale:	Integer
UpperBound:	
LowerBound:	
Interval:	

Name:	Percentage
TypeOfScale:	Double
UpperBound:	1.0
LowerBound:	0.0
Interval:	

Name:	Boolean
TypeOfScale:	Boolean
Values:	

Name:	Schulnoten
TypeOfScale:	String
Values:	sehr gut, gut, befriedigend, ausreichend, mangelhaft, ungenügend

Name:	HighLow
TypeOfScale:	String
Values:	Very High, High, Middle, Low, Very Low

B.2 Auxiliary Functions

B.2.1 Property level functions

NAME:	acv(c)
INPUTPARAMETERS:	a Class
OUPTVALUETYPE:	Boolean
USEDINMODELELEMENT:	Predicate that indicates if a given class can access the attribute
INFORMALDEFINITION:	
OCLDEFINITION:	**context** Property def:acv(c:Class): **Boolean** = ((self.class.oclAsType(Class) .oclIsKindOf(OclInvalid) = false) **and** (self.class = c)) **or** (self.visibility = VisibilityKind::public) **or** (self.visibility = VisibilityKind::protected) **and** self.class.descendants()->includes(c)
COMMENTS:	[eA01]
LITERATURS:	

B.2.2 Class level functions

NAME:	children()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of directly derived classes
OCLDEFINITION:	**context** Class def: children():**Set**(Class) = Class.allInstances()->select(c| c.parents()->includes(self))
COMMENTS:	
LITERATURS:	[eA01]

NAME:	descendants()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of all derived classes (either directly or indirectly)
OCLDEFINITION:	**context** Class def:descendants(): **Set**(Class) = children()-> iterate(elem: Class; acc: **Set**(Class)=children() | acc-> union (elem.descendants()))
COMMENTS:	This operation is recursive. Notice that even with multiple inheritance the result is a set (no repeated classes)
LITERATURS:	[eA01]

NAME:	parents()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of classes from which the current class derives directly
OCLDEFINITION:	predefined in UML/Z Meta Model
COMMENTS:	
LITERATURS:	[eA01]

NAME:	ascendants()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of classes from which the current class derives directly or indirectly
OCLDEFINITION:	**context** Class def:ascendants(): **Set**(Class) = parents()-> iterate(elem: Class; acc: **Set**(Class)=parents() | acc-> union (elem.ascendants()))
COMMENTS:	This operation is recursive. Notice that even with common ancestors due to multiple inheritance the result is a set (no repeated classes)
LITERATURS:	[eA01]

NAME:	coupledClasses()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of classes to which the current class is coupled (excluding inheritance)
OCLDEFINITION:	**context** Class def: coupledClasses(): **Set**(Classifier) = self.getAllAttributes().type .oclAsType(Classifier)->asSet()->union(self.ownedOperation.getType() .oclAsType(Classifier) ->asSet()->union(self.ownedOperation.ownedParameter.type .oclAsType(Classifier)->asSet()))
COMMENTS:	Shortened according to UMLZ model This function includes the coupled classes corresponding to: return type of class operations and type of class parameters
LITERATURS:	[eA01]

NAME:	newOperations()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Operation)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Operations defined in the class that are not overriding inherited ones
OCLDEFINITION:	**context** Class def:newOperations(): **Set**(Operation) = definedOperations() - inheritedOperations()
COMMENTS:	The definition excludes inherited Operations (and consequently, it excludes overridden Operations)
LITERATURS:	[eA01]

NAME:	definedOperations()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Operation)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of operations defined (not inherited) in the class

OCLDEFINITION:	`context Class def:definedOperations():`
	`  Set(Operation)`
	`  = getOperations()`

COMMENTS:	
LITERATURS:	[eA01]

NAME:	overriddenOperations()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Operation)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of operations defined in the class that override inherited ones = set of inherited operations that are overridden by locally defined ones

OCLDEFINITION:	`context Class def:overriddenOperations():`
	`  Set(Operation)=`
	`  if (definedOperations()->size() > 0 and`
	`  inheritedOperations()->size()>0) then`
	`  definedOperations()-inheritedOperations()`
	`  else Set{} endif`

COMMENTS:	
LITERATURS:	[eA01]

NAME:	getAllOperations()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Operation)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of all the Operations of this class, including those inherited from its parents
OCLDEFINITION:	The OCL–Query is predefined in the UML/Z Meta model
COMMENTS:	
LITERATURS:	[BeA03a]

NAME:	newAttributes()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Property)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Attributes defined in the class that are not overriding inherited ones
OCLDEFINITION:	**context** Class def:newAttributes(): **Set**(Property) = definedAttributes() – inheritedAttributes()
COMMENTS:	The definition excludes inherited Attributes (and consequently, it excludes overridden Attributes)
LITERATURS:	[eA01]

NAME:	inheritedAttributes()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Property)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of inherited attributes that are not overridden by locally defined ones
OCLDEFINITION:	**context** Class def:inheritedAttributes(): **Set**(Property) = self.getInheritedMembers()–> select(f:Member|f.oclIsKindOf(Property)) –>collect(f|f.oclAsType(Property))–> asSet()
COMMENTS:	
LITERATURS:	[eA01]

NAME:	definedAttributes()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Property)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of attributes defined (not inherited) in the class
OCLDEFINITION:	**context** Class def:definedAttributes(): **Set**(Property) = self.attribute->asSet()
COMMENTS:	
LITERATURS:	[eA01]

NAME:	overriddenAttributes()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Property)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of attributes defined in the class that override inherited ones = Set of inherited attributes that are overridden by locally defined ones
OCLDEFINITION:	**context** Class def:overriddenAttributes(): **Set**(Property) = **if** (definedAttributes()->size() > 0 **and** inheritedAttributes()->size()>0) **then** definedAttributes()-inheritedAttributes() **else Set**{} **endif**
COMMENTS:	
LITERATURS:	[eA01]

NAME:	getAllAttributes()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Property)
USEDINMODELELEMENT:	Class
INFORMALDEFINITION:	Set of all the attributes of this class, including those inherited from its parents
OCLDEFINITION:	The OCL–Query is predefined in the UML/Z Meta model
COMMENTS:	
LITERATURS:	[eA01]

B.2.3 Model level functions

NAME:	allClasses()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Model
INFORMALDEFINITION:	Set of all classes belonging to the model
OCLDEFINITION:	`context Model def: allClasses(): Set(Class) = (Class.allInstances()->asSet())->iterate(c:Class; acc:Set(Class) = Set{}| if(c.oclIsKindOf(Behavior)) then acc else acc->including(c) endif)`
COMMENTS:	
LITERATURS:	[eA01]

NAME:	baseClasses()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(Class)
USEDINMODELELEMENT:	Model
INFORMALDEFINITION:	Set of base classes of the model that are inherited by some classes
OCLDEFINITION:	`context Model def:baseClasses(): Set(Class)= allClasses().parents().oclAsType(Class) ->asSet()`
COMMENTS:	
LITERATURS:	[eA01]

B.2.4 StateMachine level functions

NAME:	directSimpleStates()
INPUTPARAMETERS:	
OUPTVALUETYPE:	Set(State)
USEDINMODELELEMENT:	StateMachine
INFORMALDEFINITION:	Set of all simple states of a StateMachine not considering the simple states within the composites states
OCLDEFINITION:	`context StateMachine def: directSimpleStates(): Set(State) = self.region->iterate(region:Region; acc: Set(State)= Set{}|acc->union(region.subvertex->select(a | a.oclIsTypeOf(State))-> select(a:State| a.isSimple).oclAsType(State) -> asSet()))`
COMMENTS:	
LITERATURS:	[eA01]

B.3 Base Measures

B.3.1 Unit of measure is Class

NAME:	Root class predicate
ACRONYM:	IsRoot
UNITOFMEASUREMENT:	Class
SCALE:	Boolean
INFORMALDEFINITION:	Root class predicate - indicates that it has no ascendants
OCLDEFINITION:	**context** Class def:IsRoot(): **Boolean** = parents()->isEmpty()
COMMENTS:	
USINGAUXILARYFUNCTIONS:	parents()
LITERATURS:	[BeA03a]

NAME:	Leaf class predicate
ACRONYM:	IsLeaf
UNITOFMEASUREMENT:	Class
SCALE:	Boolean
INFORMALDEFINITION:	Leaf class predicate - indicates that it has no descendants
OCLDEFINITION:	**context** Class def:IsLeaf(): **Boolean** = children()->isEmpty()
COMMENTS:	
USINGAUXILARYFUNCTIONS:	children()
LITERATURS:	[eA01]

NAME:	Abstract Class Predicate
ACRONYM:	IsAbstract
UNITOFMEASUREMENT:	Class
SCALE:	Boolean
INFORMALDEFINITION:	Abstract Class Predicate - indicates that it is abstract
OCLDEFINITION:	**context** Class def:IsAbstract(): **Boolean** = self.isAbstract
COMMENTS:	
USINGAUXILARYFUNCTIONS:	children()
LITERATURS:	

NAME:	Available Attributes Number
ACRONYM:	AAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of all Available Attributes of a Class
OCLDEFINITION:	`context Class def: AAN():Integer = self.getAllAttributes()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllAttributes()
LITERATURS:	[BeA03a]

NAME:	Defined Attributes Number
ACRONYM:	DAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of the new attributes and the overridden attributes of a Class
OCLDEFINITION:	`context Class def:DAN():Integer = definedAttributes()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	definedAttributes()
LITERATURS:	[BeA03b]

NAME:	New Attributes Number
ACRONYM:	NAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of Attributes defined in the class that are not overriding inherited ones
OCLDEFINITION:	`context Class def:NAN(): Integer = newAttributes()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	newAttributes()
LITERATURS:	[BeA03a]

NAME:	Inherited Attributes Number
ACRONYM:	IAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of Attributes inherited by the class
OCLDEFINITION:	`context Class def:IAN():Integer = inheritedAttributes() -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	inheritedAttributes()
LITERATURS:	[BeA03a]

NAME:	Overridden Attributes Number
ACRONYM:	OAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of Attributes overridden by a Class
OCLDEFINITION:	`context Class def:OAN():Integer = self.overriddenAttributes()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	overriddenAttributes()
LITERATURS:	[BeA03a]

NAME:	Available Operations Number
ACRONYM:	AON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of all available operations of a Class
OCLDEFINITION:	`context Class def:AON():Integer = self.getAllOperations()-> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllOperations()
LITERATURS:	[BeA03a], [BPC05]

NAME:	Defined Operations Number
ACRONYM:	DON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of the new operations and the overridden operations of a Class
OCLDEFINITION:	`context Class def:DON():Integer = definedOperations()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	definedOperations()
LITERATURS:	[BeA03b]

NAME:	Inherited Operations Number
ACRONYM:	ION
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of operations inherited by the class
OCLDEFINITION:	`context Class def:ION():Integer = inheritedOperations() -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	inheritedOperations()
LITERATURS:	[BeA03a]

NAME:	Overridden Operations Number
ACRONYM:	OON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of operations overridden by a Class
OCLDEFINITION:	`context Class def:OON():Integer = overriddenOperations()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	overriddenOperations()
LITERATURS:	[BeA03a]

NAME:	New Operations Number
ACRONYM:	NON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of operations defined in the class that are not overriding inherited ones
OCLDEFINITION:	`context Class def:NON(): Integer = newOperations()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	newOperations()
LITERATURS:	[eA01], [BeA03a]

NAME:	New Operations Number
ACRONYM:	NON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of operations defined in the class that are not overriding inherited ones
OCLDEFINITION:	`context Class def:NON(): Integer = newOperations()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	newOperations()
LITERATURS:	[eA01], [BeA03a]

NAME:	Protected Attributes Number
ACRONYM:	PROAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of Protected Attributes in the Class
OCLDEFINITION:	`context Class def:PROAN(): Integer = self.getAllAttributes() -> select( a | a.visibility = VisibilityKind::protected) -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllAttributes()
LITERATURS:	[BeA03a]

NAME:	Public Attributes Number
ACRONYM:	PUBAN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of public Attributes in the Class
OCLDEFINITION:	`context Class def:PUBAN(): Integer = self.getAllAttributes() -> select( a | a.visibility = VisibilityKind::public) -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllAttributes()
LITERATURS:	[BeA03a]

NAME:	Private Operations Number
ACRONYM:	PRION
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of private Operations in the Class
OCLDEFINITION:	`context Class def:PRION(): Integer = self.getAllOperations() -> select( a | a.visibility = VisibilityKind::private) -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllOperations()
LITERATURS:	[BeA03a]

NAME:	Protected Operations Number
ACRONYM:	PROON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of protected Operations in the Class
OCLDEFINITION:	`context Class def:PROON(): Integer = self.getAllOperations() -> select( a | a.visibility = VisibilityKind::protected) -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllOperations()
LITERATURS:	[BeA03a]

NAME:	Public Operations Number
ACRONYM:	PUBON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of public Operations in the Class
OCLDEFINITION:	`context Class def:PUBON(): Integer = self.getAllOperations() -> select( a |a.visibility = VisibilityKind::public) -> size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	getAllOperations()
LITERATURS:	[BeA03a]

NAME:	Children Number
ACRONYM:	CHIN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of directly derived Classes of the Class
OCLDEFINITION:	`context Class def:CHIN(): Integer = children()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	children()
LITERATURS:	[eA01], [BeA03a]

NAME:	Descendants Number
ACRONYM:	DESN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of all derived classes (either directly or indirectly)
OCLDEFINITION:	`context Class def:DESN(): Integer = descendants()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	descendants()
LITERATURS:	[eA01], [BeA03a]

NAME:	Parents Number
ACRONYM:	PARN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of classes from which the current class derives directly
OCLDEFINITION:	`context Class def:PARN(): Integer = parents()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	parents()
LITERATURS:	[eA01], [BeA03a]

NAME:	Ascendants Number
ACRONYM:	ASCN
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of classes from which the current class derives directly or indirectly
OCLDEFINITION:	`context Class def:ASCN(): Integer = ascendants()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	ascendants()
LITERATURS:	[eA01], [BeA03a]

NAME:	Class Name Length
ACRONYM:	CNL
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Length of the name of a Class
OCLDEFINITION:	`context Class def:CNL():Integer = self.name.size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
LITERATURS:	

B.3.2 Unit of measure is Model

NAME:	Unnamed Associations Number
ACRONYM:	UnAssocN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Number of unnamed Associations in a Model
OCLDEFINITION:	`context Model def: UnAssocN():Integer = Association.allInstances()->iterate(a: Association; acc:Integer = 0| if(( a.name.size() = 0 or a.name.size().oclIsTypeOf(Integer) = false)) then acc+1 else acc+0 endif)`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
LITERATURS:	

NAME:	Total Classes Number
ACRONYM:	TCN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of classes in the Model
OCLDEFINITION:	`context Model def:TCN(): Integer = allClasses()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	AllClasses()
LITERATURS:	[eA01]

NAME:	Number of Associations
ACRONYM:	NAssoc
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of associations in the model
OCLDEFINITION:	`context Model def:NAssoc(): Integer = Association.allInstances()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
LITERATURS:	[BPC05]

B.3.3 Unit of measure is StateMachine

NAME:	Number of Direct Simple States
ACRONYM:	NDSS
UNITOFMEASUREMENT:	StateMachine
SCALE:	Integer
INFORMALDEFINITION:	The number of simple states of a StateMachine not considering the simple states within the composites states
OCLDEFINITION:	`context StateMachine def: NDSS(): Integer = self.directSimpleStates()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
LITERATURS:	

NAME:	Number of Direct Composite States
ACRONYM:	NDCS
UNITOFMEASUREMENT:	StateMachine
SCALE:	Integer
INFORMALDEFINITION:	The number of composite states of a StateMachine not considering the composite states within the composite states of the StateMachine (StateChart)
OCLDEFINITION:	`context StateMachine def: NDCS(): Integer = self.directCompositeStates()->size()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
LITERATURS:	

B.4 Derived measures

B.4.1 Unit of measure is Class

NAME:	Depth of Inheritance of a class
ACRONYM:	DIT
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	The Depth of Inheritance of a class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.
OCLDEFINITION:	`context Class def:DIT():Integer = (if self.IsRoot() then 0 else 1 + self.parents()-> iterate(elem:Class; acc:Integer=0 | if elem.DIT() > acc then elem.DIT() else acc endif) endif)`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Root class predicate (IsRoot)
LITERATURS:	[BeA03a]

NAME:	Unnecessary Hierarchy
ACRONYM:	UnHy
UNITOFMEASUREMENT:	Class
SCALE:	Boolean
INFORMALDEFINITION:	lass Hierarchy is unnecessary, if class is abstract and only one direct subclass available.
OCLDEFINITION:	`context Class def: UnHy(): Boolean = self.IsAbstract() and self.DDESN() = 1`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Abstract Class Predicate (isAbstract), Direct Descendants Number (DDESN)
LITERATURS:	

NAME:	Small Classes
ACRONYM:	SmallC
UNITOFMEASUREMENT:	Class
SCALE:	Percentage
INFORMALDEFINITION:	A class is a small class if the total number of its operations and its attributes is lower than one to fourth than the average for all Classes.
OCLDEFINITION:	`context Class def:SmallC (): Boolean = if ((self.DAN()+self.DON()) / ((self.getModel().TDAN() + self.getModel().TDON()) / self.getModel().TCN()) < 0.33) then true else false endif`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Defined Operations Number (DON), Defined Attributes Number (DAN), Total Defined Attributes Number (TDAN), Total Defined Operations Number (TDON), Total Classes Number (TCN)
LITERATURS:	

NAME:	Public Attributes Ratio
ACRONYM:	RPubA
UNITOFMEASUREMENT:	Class
SCALE:	Percentage
INFORMALDEFINITION:	Divide public attributes by all attributes of a class.
OCLDEFINITION:	**context** Class def: RPubA(): **Real** = PUBAN() / AAN() **inv**: AAN() != 0
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Public Attributes Number (PUBAN), Available Attributes Number (AAN)
LITERATURS:	

NAME:	Defined Operations Number
ACRONYM:	DON
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of the new operations and the overridden operations of a Class
OCLDEFINITION:	**context** Class def:DON():**Integer** = definedOperations()->size()
COMMENTS:	
USINGAUXILARYFUNCTIONS:	definedOperations()
USINGMEASURES:	
LITERATURS:	[BeA03b]

NAME:	Size of local Attributes and operations
ACRONYM:	SIZE2
UNITOFMEASUREMENT:	Class
SCALE:	Integer
INFORMALDEFINITION:	Number of Attributes + Number of local Operations
OCLDEFINITION:	`context Class def: SIZE2(): Integer = self.DON() + self.DAN()`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Defined Operations Number (DON), Defined Attributes Number (DAN)
LITERATURS:	Genero2005, [BeA03a]

B.4.2 Unit of measure is Model

NAME:	Large Classes Number
ACRONYM:	LCN
UNITOFMEASUREMENT:	Model
SCALE:	Percentage
INFORMALDEFINITION:	Total number of Large Classes in the model. A class is a Large Class if the total number of its operations and its attributes is more than three times higher than the average for all Classes.
OCLDEFINITION:	`context Model def: LCN(): Integer = allClasses() -> iterate (elem : Class ; acc : Integer = 0 | if (((elem.DON()+ elem.DAN()) / (self.TDAN()+self.TDON())) > 3) then acc + 1 else acc + 0 endif)`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Defined Operations Number (DON), Defined Attributes Number (DAN), Total Defined Attributes Number (TDAN), Total Defined Operations Number (TDON)
LITERATURS:	

NAME:	Total New Attributes Number
ACRONYM:	TNAN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of new attributes in the model.
OCLDEFINITION:	`context Model def:TNAN(): Integer = allClasses()->iterate(elem: Class; acc: Integer = 0 | acc + elem.NAN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	New Attributes Number (NAN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Defined Attributes Number
ACRONYM:	TDAN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of defined attributes in the model.
OCLDEFINITION:	`context Model def:TDAN(): Integer = allClasses()->iterate(elem: Class; acc: Integer = 0 | acc + elem.DAN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Defined Attributes Number (DAN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Inherited Attributes Number
ACRONYM:	TIAN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of inherited attributes in the model.
OCLDEFINITION:	`context Model def:TIAN(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.IAN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Inherited Attributes Number (IAN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Overridden Attributes Number
ACRONYM:	TOAN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of overridden attributes in the model.
OCLDEFINITION:	`context Model def:TOAN(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.OAN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Overridden Attributes Number (OAN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Available Attributes Number
ACRONYM:	TAAN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of available attributes in the model.
OCLDEFINITION:	`context Model def:TAAN(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.AAN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Available Attributes Number (AAN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total New Operations Number
ACRONYM:	TNON
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of new operations in the model.
OCLDEFINITION:	`context Model def:TNON(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.NON())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	New Operations Number (NON)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Defined Operations Number
ACRONYM:	TDON
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of operations defined in the model.
OCLDEFINITION:	`context Model def:TDON(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.DON())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Defined Operations Number (DON)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Inherited Operations Number
ACRONYM:	TION
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of inherited operations in the model.
OCLDEFINITION:	`context Model def:TION(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.ION())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Inherited Operations Number (ION)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Overridden Operations Number
ACRONYM:	TOON
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of overridden operations in the model.
OCLDEFINITION:	`context Model def:TOON(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.OON())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Overridden Operations Number (OON)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Available Operations Number
ACRONYM:	TAON
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of available operations in the model.
OCLDEFINITION:	`context Model def:TAON(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.AON())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Available Operations Number (AON)
LITERATURS:	[eA01], [BeA03a]

NAME:	Total Inheritance Links Number
ACRONYM:	TILN
UNITOFMEASUREMENT:	Model
SCALE:	Integer
INFORMALDEFINITION:	Total number of inheritance relations in the model.
OCLDEFINITION:	`context Model def:TILN(): Integer = allClasses()-> iterate(elem: Class; acc: Integer = 0 | acc + elem.PARN())`
COMMENTS:	
USINGAUXILARYFUNCTIONS:	allClasses()
USINGMEASURES:	Parents Number (PARN)
LITERATURS:	[eA01], [BeA03a]

NAME:	Model Attribute Inheritance Factor
ACRONYM:	MAIF
UNITOFMEASUREMENT:	Model
SCALE:	Percentage
INFORMALDEFINITION:	The Attribute Inheritance Factor is defined as a quotient between the sum of inherited attributes in all classes of the system under consideration and the total number of available attributes (locally defined plus inherited) for all classes.
OCLDEFINITION:	**context** Model def: MAIF(): **Real** = self.TIAN() / self.TAAN() **inv**: self.TAAN() > 0
COMMENTS:	MOOD metric set
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Total Inherited Attributes Number (TIAN), Total Available Attributes Number (TAAN)
LITERATURS:	Genero2005, [BeA03a]

NAME:	Model Operations Inheritance Factor
ACRONYM:	MOIF
UNITOFMEASUREMENT:	Model
SCALE:	Percentage
INFORMALDEFINITION:	The Operations Inheritance Factor is defined as a quotient between the sum of inherited operations in all classes of the system under consideration and the total number of available operations (locally defined plus inherited) for all classes.
OCLDEFINITION:	**context** Model def: MOIF(): **Real** = self.TION() / self.TAON() **inv**: self.TAON() > 0
COMMENTS:	MOOD metric set
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Total Inherited Operations Number (TION), Total Available Operations Number (TAON)
LITERATURS:	Genero2005, [BeA03a]

NAME:	Polymorphism Factor
ACRONYM:	POF
UNITOFMEASUREMENT:	Model
SCALE:	Percentage
INFORMALDEFINITION:	Quotient between the actual number of possible different polymorphic situations and the maximum number of possible distinct polymorphic situations (due to inheritance).
OCLDEFINITION:	**context** Model def: POF(): **Real** = TOON() / TAON() **inv**: TAON() > 0
COMMENTS:	MOOD metric set
USINGAUXILARYFUNCTIONS:	
USINGMEASURES:	Total Available Operations Number (TAON), Total Overridden Operations Number (TOON)
LITERATURS:	Genero2005, [BeA03a]

B.5 Indicators

B.5.1 Unit of measure is Class

NAME:	Public Attributes
ACRONYM:	PubA
UNITOFMEASUREMENT:	Class
SCALE:	Percentage
CALCULATION:	IF MAHF() = 1 THEN SEHR GUT IF MAHF() > 0.9 THEN GUT IF MAHF() > 0.8 THEN BEFRIEDIGEND IF MAHF() > 0.7 THEN AUSREICHEND IF MAHF() > 0.6 THEN MANGELHAFT ELSE UNGENÜGEND
COMMENTS:	Fuzzy Logic
USINGMEASURES:	
LITERATURS:	

NAME:	Small Classes
ACRONYM:	I_SmallC
UNITOFMEASUREMENT:	Class
SCALE:	Boolean
CALCULATION:	IF SmallC() == true THEN true IF SmallC() == false THEN false ELSE false
COMMENTS:	Fuzzy Logic
USINGMEASURES:	Small Class (SmallC)
LITERATURS:	

B.5.2 Unit of measure is Model

NAME:	Large Classes
ACRONYM:	I_LC
UNITOFMEASUREMENT:	Model
SCALE:	HighLow
CALCULATION:	IF LCN() > 4 THEN VERY HIGH IF LCN() > 3 THEN HIGH IF LCN() > 2 THEN MIDDLE IF LCN() > 1 THEN LOW ELSE VERY LOW
COMMENTS:	Fuzzy Logic
USINGMEASURES:	Large Classes Number (LCN)
LITERATURS:	

NAME:	Unnamed Associations
ACRONYM:	I_UnAssocs
UNITOFMEASUREMENT:	Model
SCALE:	HighLow
CALCULATION:	IF UnAssocN() / NAssoc() > 0.5 THEN VERY HIGH IF UnAssocN() / NAssoc() > 0.3 THEN HIGH IF UnAssocN() / NAssoc() > 0.1 THEN MIDDLE IF UnAssocN() / NAssoc() > 0.05 THEN LOW ELSE VERY LOW
COMMENTS:	Fuzzy Logic
USINGMEASURES:	Unnamed Associations Number (UnAssocN), Number of Associations (NAssoc)
LITERATURS:	

NAME:	Short Class Names
ACRONYM:	I_SCN
UNITOFMEASUREMENT:	Class
SCALE:	HighLow
CALCULATION:	IF CNL() < 2 THEN VERY HIGH IF CNL() >= 2 && CNL() < 3 THEN HIGH IF CNL() >= 3 && CNL() < 5 THEN MIDDLE IF CNL() >= 5 && CNL() < 8 THEN LOW ELSE VERY LOW
COMMENTS:	Fuzzy Logic
USINGMEASURES:	Class Name Length (CNL)
LITERATURS:	

C Lizenz

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not

the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a

warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the

same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the

Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.